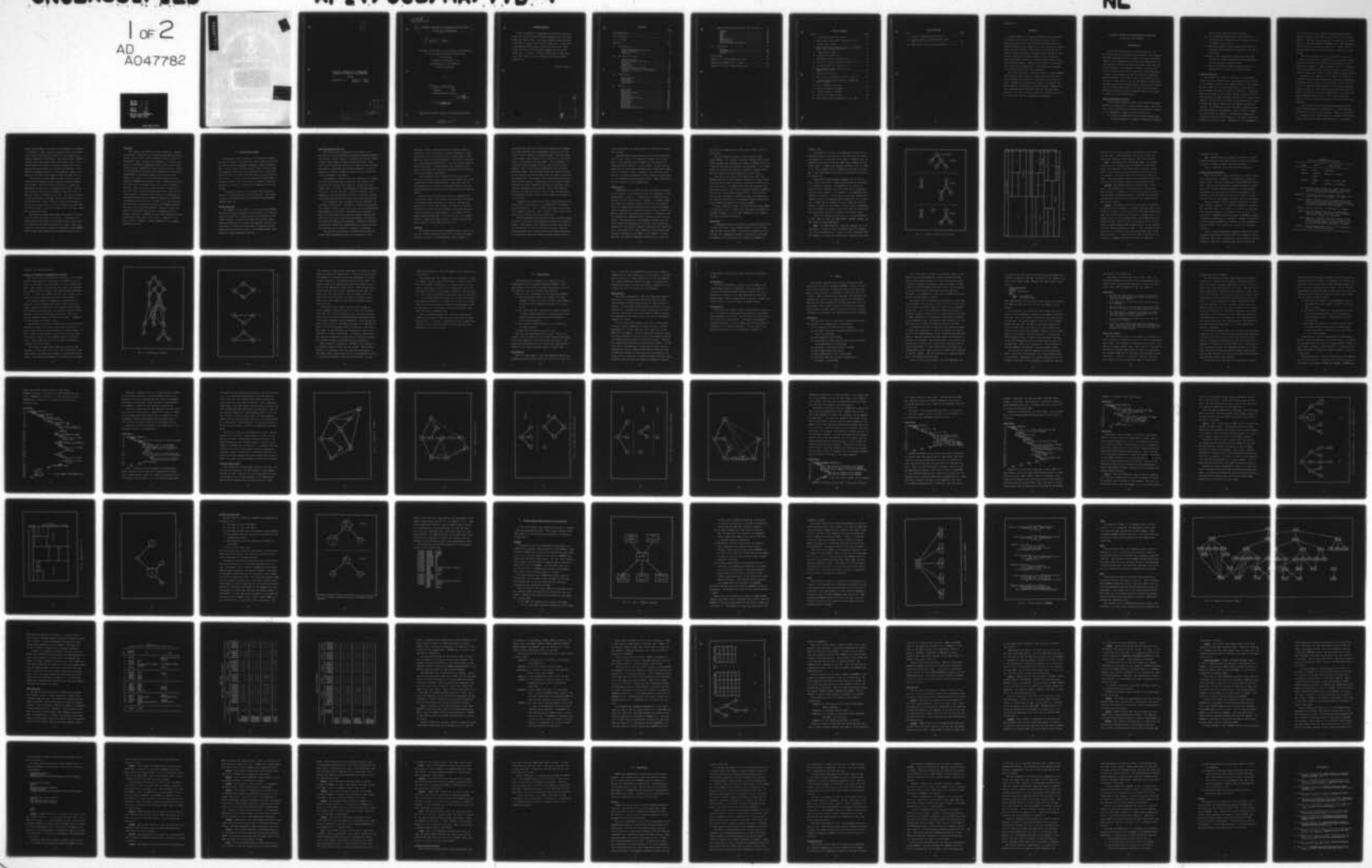
AD-A047 782 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
AUTOMATIC GENERATION OF SEGMENTATION DIRECTIVES ON THE CDC 6600--ETC(U)
DEC 77 S R SARNER
UNCLASSIFIED AFIT/GCS/MA/77D-4 NL

1 OF 2
AD
A047782



(1)

AUTOMATIC GENERATION OF SEGMENTATION
DIRECTIVES ON THE CDC 6600 COMPUTER

THESIS

GCS/MA/77D-4 Steven R. Sarner
Captain USAF

DDC
DEC 21 1977
A

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

(14) AFIT/
GCS/MA/77D-4

(6) AUTOMATIC GENERATION OF SEGMENTATION DIRECTIVES
ON THE CDC 6600 COMPUTER.

(9) Master's THESIS,

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

(10) Steven R. Sarner, B.S.
Captain USAF
Graduate Computer Science

(12) 110p.

(11) December 1977

Approved for public release; distribution unlimited

1473
012 225

LB

Acknowledgements

I wish to express my indebtedness to my thesis adviser, Charles W. Richard, whose off-hand remark was the seed from which this thesis grew. His timely comments helped me many times when I thought I had reached a dead end. I also wish to express my appreciation to Captains George Orr, Chuck Roark, and Pete Miller for their suggestions to my very rough draft. Finally, I wish to acknowledge my gratitude to my wife, Suzie, for not jumping on me too much when I ignored her.

Steven R. Sarner

Contents

List of Figures

List of Tables

Abstract

Central memory is a critical resource on the Aeronautical Systems Division CDC 6600 computer system. This is especially true in the time-sharing environment where the standard maximum field length for a user's program is 60000_8 words of central memory. The amount of central memory required to load and execute a program may be reduced by using the CDC segmentation scheme in which a tree structure of relocatable object modules is specified which permits sharing of central memory. However, the design of an effective tree structure for segmenting a large application program can be very difficult and time consuming.

This thesis develops a software processor called SEGMENT that automatically generates segmentation directives for a user's program that describe a tree structure for that program. SEGMENT uses a depth-first search to determine the tree structure of relocatable object modules. SEGMENT also uses algorithms which correctly position labeled common blocks in this tree structure and which eliminate data preset errors and indirect references to externals.

AUTOMATIC GENERATION OF SEGMENTATION DIRECTIVES
ON THE CDC 6600 COMPUTER

I. Introduction

Since the earliest days of digital computers, the relatively small, high speed central memory of the computer has been augmented by a much larger, but slower, secondary memory. This arrangement facilitates multiprogramming and also makes it possible to execute programs larger than the central memory of the computer. However, the use of two levels of memory brings up the problem of what information to store in each memory and when to move the information between memories. More specifically, this problem can be stated as: when should a unit of information be moved from secondary memory into central memory, into what unallocated area of central memory should this information be placed, and what information should be removed from central memory if there is no unallocated space.

Memory Management Solutions

This problem, generally known as the memory management problem, has been solved in many different ways. Each solution can be classified by the level of abstraction at which it is implemented. Several classifications are:

(1) Paging and segmented virtual memory systems where memory management is handled by dedicated hardware

and low level operating system routines.

(2) Job scheduling systems where each job step of a job causes some information to be transferred from one memory to the other.

(3) Operating system utility processors that move information between memories according to the user's directives.

(4) Higher order languages with features that allow the user to control information flow between memories at the source language level.

(5) Application programs where information flow is controlled by the user at the program level.

Specific Solution

This thesis will discuss a solution at the operating system utility processor level. The processor in this case is the loader on the CDC 6600 computer (Ref 1). The purpose of this loader is to place programs into central memory in such a manner that they are ready for execution. Functions that are performed by the loader include: loading relocatable and absolute object modules, generation of overlays and segments, generation of load maps, and presetting data areas to specified values. When given the proper directives by the user, the loader will divide the user's application program into pieces called segments. After the first segment has been loaded and executed, the remaining segments are automatically moved from one memory to the other without any intervention from the user. Segments that are independent

from one another, i.e., execution cannot proceed directly from one segment to the other, do not have to be in central memory at the same time and may use the same area of central memory. Loads which generate segments are called segmented loads and allow a program to execute in a smaller area of central memory than if the program were loaded as a single entity.

The major problem with using segmented loads is that it takes an experienced programmer and a lot of time to produce the directives that specify how the program is to be broken into segments. Sayer estimates that 25 to 40 percent of total programming costs result from efforts to manually generate similar directives on other computer systems (Ref 2).

Therefore, the purpose of this paper is to describe a software processor that automatically generates segmentation directives for the loader on the CDC 6600. Due to the difficulty of defining what properties an optimally segmented program should have, the processor described here does not produce directives for optimal segmentation. It does, however, produce directives that significantly reduce the amount of central memory required for the execution of most programs.

The main difficulty in producing segmentation directives, either manually or automatically, is deciding what information to place in each segment. There are two kinds of information that can be placed in a segment: relocatable object modules and labeled common blocks. A relocatable

object module (ROM) is the basic unit produced by a compiler or assembler. A ROM consists of several tables that define blocks within the ROM, the contents of these blocks, and address relocation information. Each block within a ROM may contain either executable code or data (Ref 1:1-2). A labeled common block (LCB) is an area in memory that can be used by one or more ROMs for storage of shared data. Any ROM that references an LCB can cause specific values to be preset in the LCB by the loader at load time (Ref 1:1-2).

The decision of which ROMs and LCBs to place in each segment must be based on three relations between the ROMs and the LCBs. The first kind of relation is between two ROMs where one ROM calls or references the other. The two other kinds of relations are between a ROM and an LCB. A ROM can contain a reference to an LCB. The data in the LCB is then accessible to the ROM. That is, the ROM may use or modify this data. Also, a ROM may contain instructions to the loader to preset data in an LCB to specific values when the LCB is loaded. In order to preset data in an LCB, a ROM must reference the LCB.

Any method of generating segmentation loader directives must take these relations into account. Therefore, this paper presents algorithms for analyzing these relations and for producing directives consistent with them. These algorithms are then used to develop the processor called SEGMENT, which automatically generates the segmentation directives.

Overview

This paper is divided into several chapters. Chapter II presents a description of the memory management policies used on the CDC 6600. Three different kinds of loads are discussed: basic, overlay, and segmented. A heuristic approach to generating segmentation directives is then given and the various segmentation directives are defined. Several bad programming techniques for communicating information between ROMs in different segments are also discussed. Chapter III presents the system specifications that the processor SEGMENT was designed to meet. Chapter IV outlines the algorithms and concepts that can be used to segment a program. These algorithms include methods of analysis of the three kinds of relations described above. They also include methods of obtaining these relations, techniques for reducing execution time of segmented programs, and methods of producing the directives. Chapter V is a description of the two control card procedures and four FORTRAN programs that make up the processor SEGMENT. Chapter VI gives the results of this study and presents recommendations for follow-on projects. Appendix A gives the definitions of the graph theory terms used in this thesis. Appendix B is a user's guide to SEGMENT. Finally, Appendix C is a sample segmented load map.

II. The CDC 6600 Loader

The purpose of this chapter is to describe the operation of the loader on the CDC 6600. This loader can be used to solve the memory management problem in three different ways. Each of these solutions, basic loads, overlay loads, and segmented loads, will be discussed in turn. Then, the syntax of the five loader directives will be given and several rules for communication of information between ROMs in different segments will be outlined. Finally, a heuristic approach for breaking a program into segments will be discussed.

The specific release of the loader used during this study is the Loader Version 1.0-414L (Ref 1). This loader is part of the Network Operating System/Batch Environment (NOS/BE) (Ref 3).

Loader Functions

The purpose of the loader is to place user and NOS/BE programs into central memory in such a manner that they are ready for execution. Functions that are performed by the loader include: loading relocatable and absolute object modules, generation of overlays and segments, generation of load maps, and presetting data areas to specified values. These functions allow the user to run programs under three different memory management policies.

Memory Management Policies

The overall memory management policy used on the CDC 6600 is called relocatable partitioned memory management (Ref 4:124). Central memory is divided into several movable, variable sized blocks of words called partitions. Each user's job is given one partition, in which the job can load and execute relocatable and absolute programs. A job is a sequence of control statements, which in turn are instructions to NOS/BE or the loader.

The address of the first word in a partition is called the relative address (RA) of the job. The length of the partition and the partition itself are called the field length (FL) of the job. All addresses in programs loaded in the FL of a job are relative to the RA of the job. The hardware in the machine adds the value of the RA to any address before performing any calculations with that address.

The loader can load two kinds of object programs: relocatable and absolute. The ROMs of a relocatable program are loaded in the FL in locations that are not predetermined. Each address in each ROM is then modified (relocated) to show its relative position from some address (usually the RA) in the FL. Also, any external references to addresses not within the ROM must be satisfied. Modules of absolute programs are loaded into predetermined locations of the FL. No relocation or satisfying of externals is necessary.

The user of each partition determines which specific loader memory management policy to use within his own

partition. This is done by placing the proper loader instructions in the job stream of control statements. These instructions are in the form of one or more control statements and are called load sequences. Each control statement in a load sequence performs one of the functions (or part of one) listed above. A load sequence must terminate with a control statement that specifies the execution or non-execution of the program just loaded.

Load sequences can be thought of as the job steps of the second abstract level in the previous chapter that cause information to move between modules. When more than one load sequence is in a job, the programs loaded by each succeeding program will be able to use the same area of central memory.

Loads that do not generate overlays or segments are called basic loads. Basic loads can load either relocatable or absolute programs and they can generate load maps and preset data. For all basic loads, the loader first loads the parts of itself that are necessary to perform the functions required by the load sequence. The loader uses the highest numbered addresses in the jobs FL for this purpose. After the loader has loaded itself, it loads the specified object modules in the lower part of the FL.

<u>Overlays</u>

The second kind of memory management policy that can be used to control information flow and to increase the apparent size of the FL is called Overlays (Ref 1: Chap. 4).

Overlays allow the user to break his program into independent pieces and have these independent pieces use the same area of central memory at different times. The directives to the loader that accomplish this are placed in the source code of the program to be overlayed. This corresponds to the higher order language abstract level of the previous chapter. This source code is compiled and the resulting ROMs are loaded. The first ROM loaded must begin with one of the overlay directives which tells the loader to perform an overlay load. An overlay directive precedes each set of ROMs for that overlay. During this load, the ROMs of each overlay are loaded, address relocation is performed and all external references are satisfied. These, now absolute, overlays are then written to a file as they are generated, one record per overlay, which can be loaded and executed at a later time.

Therefore, two separate functions are performed by the loader when it processes a file of ROMs with interspersed overlay directives. First, it does an overlay load to generate absolute overlays. Then, it does a basic load of the main (first) overlay and starts execution.

This main overlay also contains a small resident loader which will load all of the remaining overlays once execution begins. Each request for an overlay load that was placed in the source code is still in the absolute code. When the program reaches one of these instructions during execution, the instruction tells the resident loader which overlay to

load next and what file that overlay is located on in secondary storage.

Overlays have several limitations that make them difficult to use. First, the loading of overlays is not automatic. The user must place requests in the source code specifying which overlay is to be loaded. Second, all communication of data between overlays must be done with common blocks or external files. Parameter lists cannot be used. Finally, all memory references (calls) between overlays must be in a downward direction toward the RA. These limitations can be overcome by using Segmentation.

Segmentation

Segmentation also allows the user to break his program up into pieces, called segments, that are independent of one another (Ref 1: Chap. 5). Segmentation is different from Overlays in that the loading of segments is automatic where the loading of overlays is done on explicit user requests. Also, the segmentation directives for dividing the program are on a separate file instead of being on the file with the ROMs as they are with overlay directives. A segmented load is signified by a SEGLOAD control statement in the load sequence. The parameters of the SEGLOAD statement tell the loader where the Segmentation directives can be found and where to put the absolute segments produced by the segmented load. The loader then loads, relocates, and satisfies the external references of the ROMs in each segment. As with Overlays, the absolute segments are written to a file and

and the first segment may be loaded with a basic load at a later time.

The root segment contains a small resident loader called SEGRES. All external references between ROMs in different segments are replaced with traps to SEGRES during the segmented load when the segments are generated. It is these traps that provide the automatic loading of segments after the root segment has been loaded with a basic load.

When one of the traps is reached during execution, SEGRES loads the correct segment, replaces the trap instruction with the original reference, and restarts execution at that instruction. If SEGRES finds that the requested segment would overwrite one or more segments already in memory, then it must unload these segments first. This involves writing all saved LCBs to secondary storage and replacing all external references to the segments being unloaded from other segments in memory with traps to SEGRES.

After a period of time, it is possible that when SEGRES is servicing a trap, it finds that the requested segment is already in memory. In this case, SEGRES will just replace the trap and restart execution.

Directives

A segmented program is made up of segments organized in a forest structure. Each segment consists of one or more ROMs and zero or more LCBs. The forest of segments may be divided into independent regions, called levels, such that no segment in one level can ever overwrite a segment in

another level.

Two segments are said to be compatible if they are in different levels or if they are in the same tree and one of the segments is an ancestor of the other. Segments that are compatible can never occupy the same area in memory at different times. A ROM in one segment may make a call to a ROM in any other segment that is compatible with the first segment. Any two segments that are not compatible are said to be conflicting.

These relationships between segments can be seen in Fig. 1 which shows how a sample program might be structured as a forest of segments. One segment of the forest is called the root segment, A in this case, and is compatible with all other segments. For example, A is compatible with segments B through T. It can also be seen that segment K is compatible with all segments except segments F, G, J, and L. Fig. 2 shows how the segments of Fig. 1 might be stored in memory. A vertical line within a level in Fig. 2 represents that the segments on either side of the line are conflicting and may use the same part of memory at different times.

Forests of segments, such as Fig. 1, can be described to the loader by using the TREE, LEVEL, INCLUDE, GLOBAL, and END directives described below.

<u>TREE</u>. The TREE directive organizes segments into tree structures. A dash is used to represent that two segments in a tree are compatible. A comma is used to represent that two segments in a tree are conflicting. Therefore, the tree

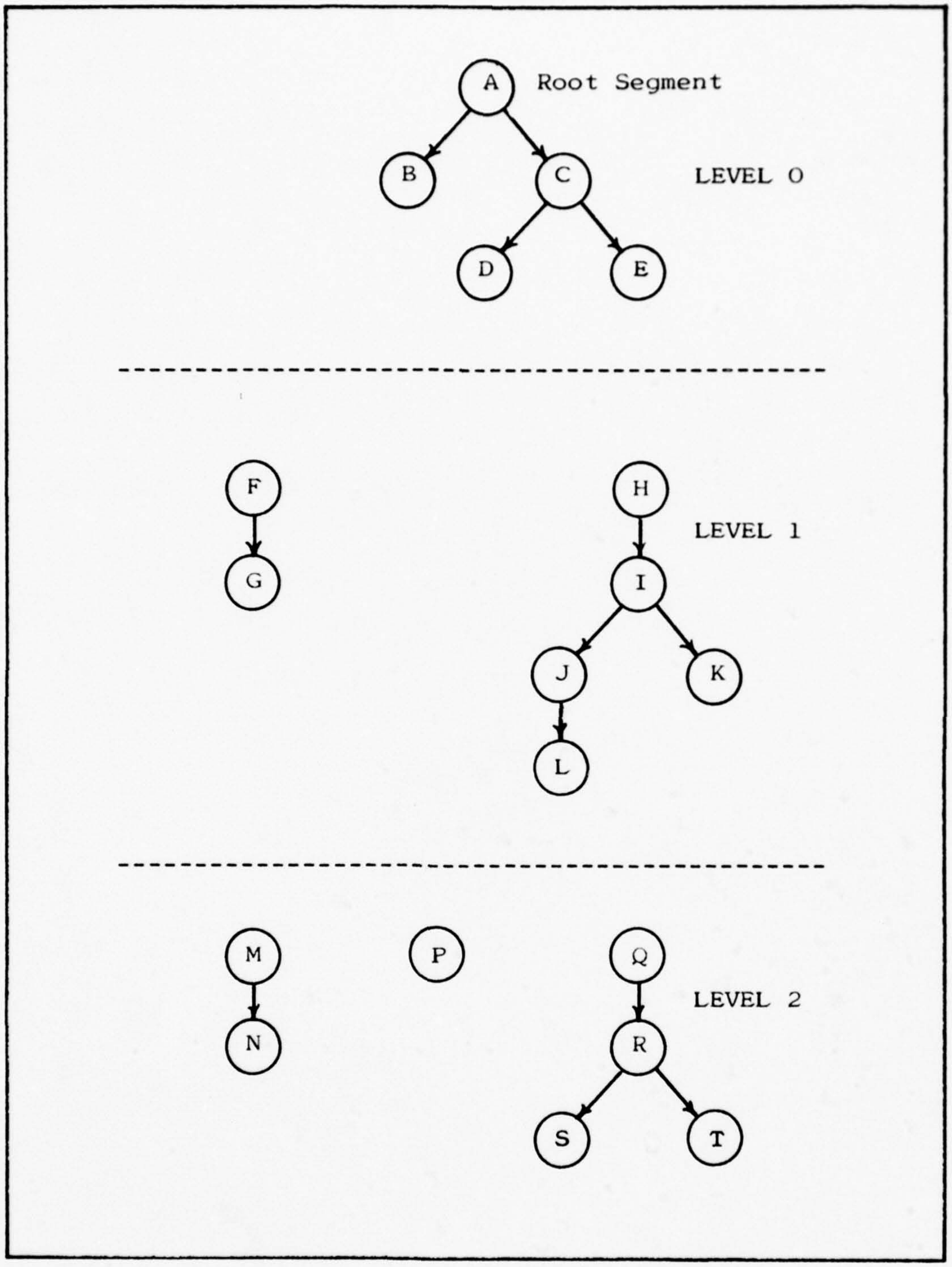


Fig. 1. A Forest Structure with Levels

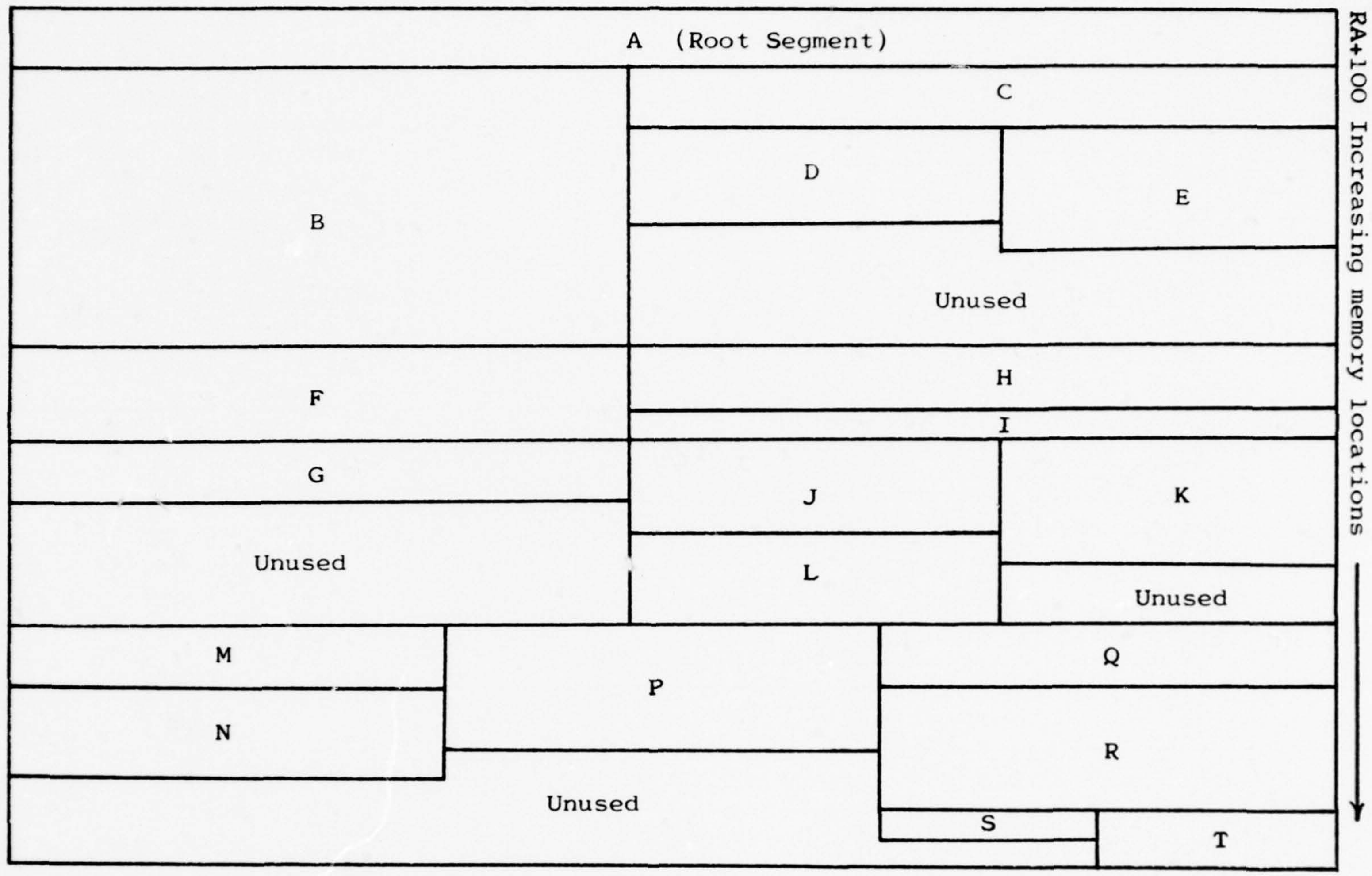


Fig. 2. Memory Map of the Forest Structure of Fig. 1

with the root Q in Fig. 1 may be represented by the expression Q-R-(S,T). The parentheses are used to delimit the conflicting segments and to separate them from the dash.

LEVEL. The LEVEL directive divides memory into compatible regions in the sense that all segments in different levels are compatible. The first level in memory, called level 0, may contain only one tree whose root must be the root segment. All other levels may contain any number of trees. The trees within a level start immediately above the last word of the highest segment in the previous level.

INCLUDE. The INCLUDE directive assigns ROMs to segments. The INCLUDE directive can be used to assign more than one ROM to a segment or to assign more than one copy of the same ROM to different segments. Any ROMs not assigned with INCLUDE directives will be placed in a compatible segment by the loader.

GLOBAL. The GLOBAL directive assigns LCBs to segments. If an LCB is not assigned to a segment, then one copy of the LCB will be placed in each segment that has ROMs referencing that LCB. An LCB should be assigned to a segment that is a common ancestor of all segments that contain ROMs which reference the LCB. The best place to assign an LCB is to the nearest common ancestor (NCA) segment of all of the segments that have ROMs referencing the LCB. If the referencing segments are in different trees, then the primary tree is the tree in level 0. It should be noted that an LCB may be assigned to a segment that does not contain a ROM which

references the LCB.

END. The END directive signifies the end of all Segmentation directives and specifies one or more entry points in the root segment at which execution may begin.

Table I gives the syntax for the five directives.

Several Frequent Problems

There are several coding techniques that inhibit effective segmentation of programs. One problem is presetting data in LCBs. This may be done with the FORTRAN DATA statement. The problem arises when a segment attempts to preset data in an LCB that has been assigned to another segment. The loader can only preset data within the segment it is loading. Since the ROM that directs the loader to preset the LCB is not in the segment, the data does not get preset.

Another problem is using indirect references to externals in other segments. An indirect reference is an external reference that is an element in a table or is the source of an external address. An example of the latter is the case where a FORTRAN sub-program passes an EXTERNAL variable to another sub-program as a parameter. Indirect references are not trapped by SEGRES and execution of the untrapped external reference will result in the use of an irrelevant address.

Both of these programming techniques should be avoided. Although programs written using these two techniques can be segmented, the amount of memory that can be saved is usually limited by their use. Unfortunately, many of the system

Table I
Syntax of Segmentation Directives

LABEL	VERB	SPECIFICATION
tname	TREE	expression
segname	INCLUDE	$\text{module}_1,\ldots,\text{module}_n$
	LEVEL	
segname	GLOBAL	$\text{bname}_1,\ldots,\text{bname}_n$-SAVE
	END	$\text{eptname}_1,\ldots,\text{eptname}_n$

tname	Optional name assigned to a tree structure by which the tree may be referenced. Tree names must be different from segment names.
expression	A character string of segment names and/or tree names linked by the operators - , (and).
segname	Name of segment in which the named object modules or labeled common blocks are to be placed. If segname is omitted, then the modules or common blocks are placed in the root segment.
module_i	Name of modules to be included in a segment.
bname_i	Names of common blocks that are addressable from any segment. May be the same as segment, entry point, program, or tree names.
-SAVE	Optional. Specifies that the contents of the global block are to be saved on a scratch file when the owning segment is overwritten and restored when the owning segmented is reloaded.
etpname_i	Optional entry point names in the root segment at which execution may begin.

libraries use these techniques.

A Heuristic Approach to Segmenting a Program

The structure of a program may be thought of as a rooted, directed graph where the vertices are the ROMs of the program and the edges are the relations, or calls, between the ROMs. Such a graph is called the call graph of the program. Fig. 3 shows the call graph of the sample program segmented in Fig. 1. Since the segments in Fig. 1 are a forest, this graph is called a Segmentation forest of the program.

The central problem of segmenting a program, then, is deriving a correct Segmentation forest from a given call graph. One way of doing this is to examine the call graph for constructs that keep the call graph from being a forest. Basically, there are three such constructs. These are cycles, redundant edges, and lattice structures. Examples of each of these are shown in Fig. 4.

The problem in each one of these cases is too many edges. Deletion of edges (2,1), (4,6), and (8,10) or (9,10) in Fig. 4 would cause all three constructs to become trees. In Fig. 3 there are two cycles: (H,I),(I,J),(J,H) and (H,I),(I,K),(K,H). There are two redundant edges: (I,Q) and (I,L). Also, there are five lattice structures ending at vertices F, H, M, P, and Q.

The easiest of these three constructs to detect and eliminate is the lattice structure. The last vertex in the lattice where the edges come together is called the lattice vertex. The heuristic technique is to break the call graph

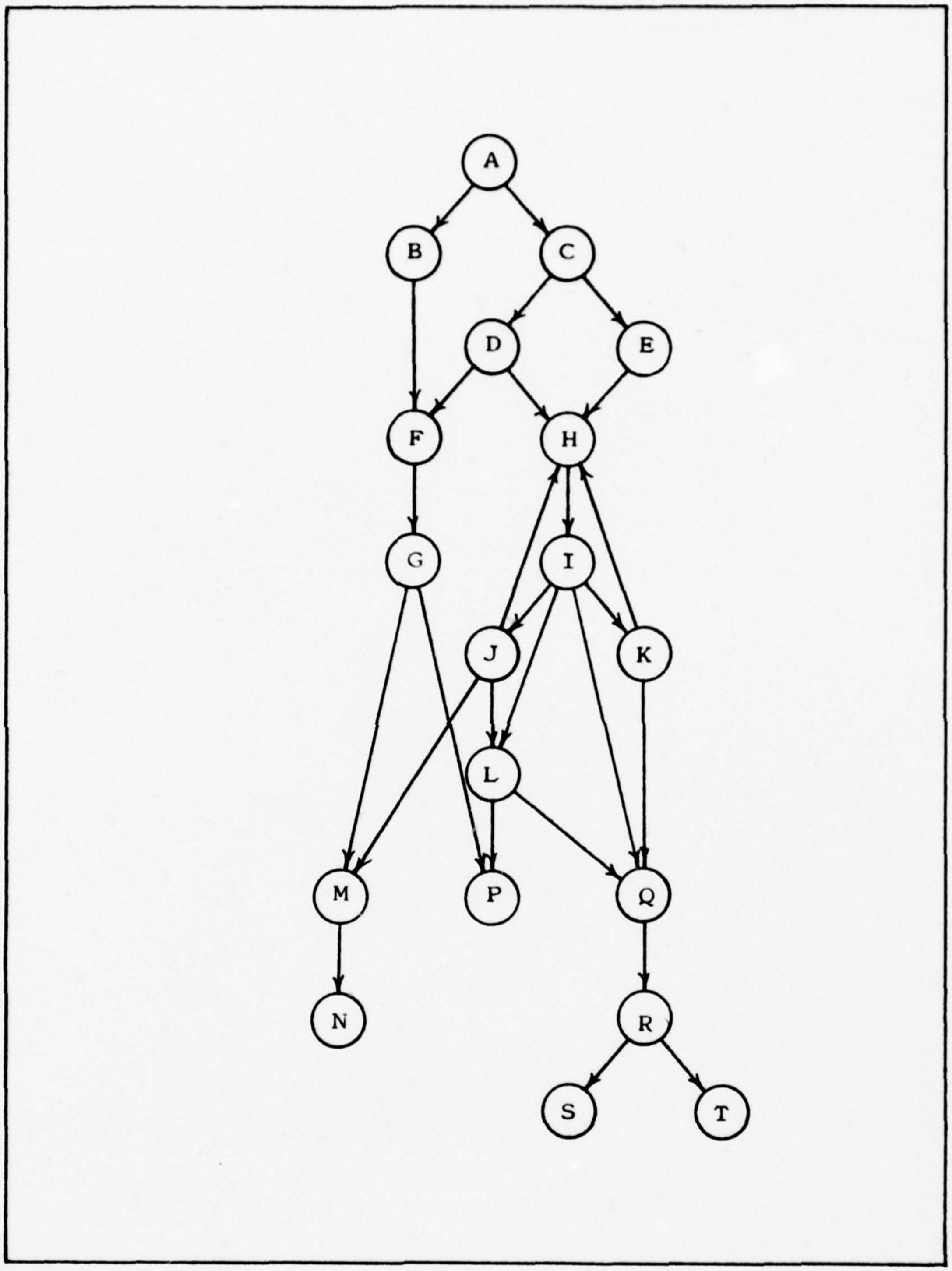


Fig. 3. A Sample Call Graph

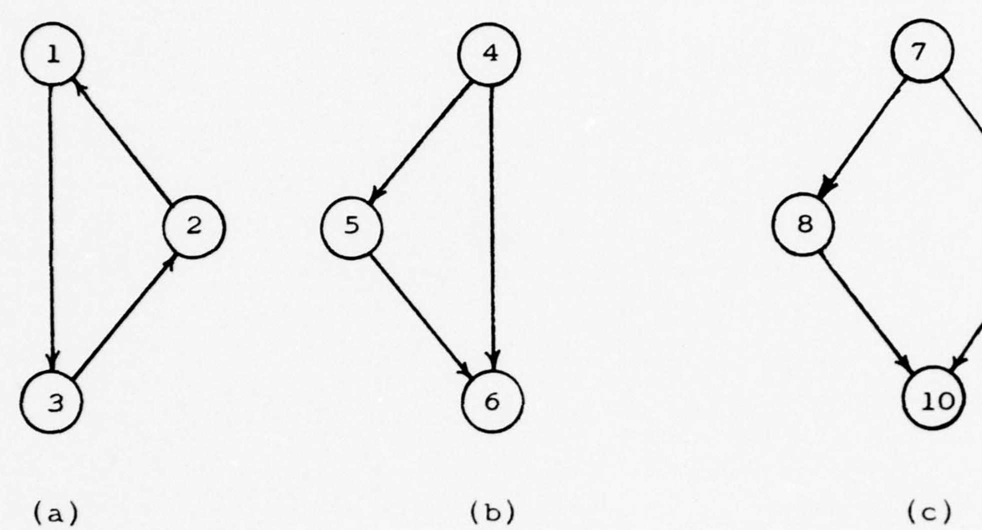


Fig. 4. Three Constructs showing a Cycle, a Redundant Edge, and a Lattice Structure

just above the lattice vertex and move all of the call graph below the break to a higher level. This can be done because the segments in different levels are compatible and, hence, can be in memory at the same time and call each other. In effect, there is an implied edge from each segment in one level to all segments in other levels. These implied edges do not mean that all of these segments actually call each other, just that they could. For this reason, when the edges (B,F), (D,F), (D,H), and (E,H) are deleted to move part of the call graph to a new level, no external references are lost.

The technique will almost give the forest shown in Fig. 1; however, there are still two cycles and one redundant edge left. So the question is now, how are the edges that cause cycles and the redundant edges detected and what is done with them when they are detected? Also, why was the edge (I,Q), in fact, a redundant edge when it is deleted as part of a lattice structure when Q is moved to a new level? The answer to the second part of the question, what to do with them is simple; delete them. In order to understand why they can be deleted and how they are detected, the method in which one ROM calls another must be understood.

When a program is executing, the ROMs can be thought of as forming a stack as they call one another. When a ROM is called, it is placed on the stack and execution begins in that ROM. When a ROM returns control to the ROM that called it, the returning ROM is taken off of the stack. At any

time during execution, all of the ROMs on the stack are said to be active.

Considering the call graph of Fig. 1, execution is begun in segment A and A is put on the stack. A calls C and C is put on the stack. Eventually, J is called and put on the stack. Now, when J calls H, H is already active and the reference to H from J can be made. Therefore, the last edge of a cycle contains redundant information and can be deleted.

In a similar manner, the fact that the top of the stack can contain the ROMs I, J, L means that I and L are compatible and I can call L with or without going through J, even if the edge (I,L) is deleted. For this reason also, the edge (I,Q) was a redundant edge.

The ideas presented here are a heuristic approach to segmenting a program. There are many other ways of accomplishing this. A specific algorithm, based on a depth-first search, will be given in Chapter IV which uses the ideas presented in this last section.

III. Requirements

The purpose of this chapter is to develop a set of requirements for the software processor SEGMENT. These requirements will be used to design the processor and also will provide a basis for testing the processor.

There are four requirements on the processor SEGMENT:

(1) The produced directives must segment the input program in such a way that its results are identical to those of the unsegmented program for all test cases.

(2) The directives must significantly reduce the amount of central memory required to load and execute most programs without imposing excessive demands on other system resources.

(3) The processor should segment all of the ROMs of the input program.

(4) The processor should be easy to use.

The first two requirements must be met. There is no reason to have a processor that produces segmented programs that do not execute or that do not save any memory. The second two requirements are not "iron-clad" requirements; however, every effort should be made to achieve them.

Requirement 1

The first requirement is that the segmented and the unsegmented programs produce identical results for every test

case. To do this, the segmentation forest must correctly represent all of the relations on the call graph. All LCBs must be placed in a common ancestor of all of the segments that contain ROMs which define the LCBs. Provisions for handling data preset in LCBs and indirect references to externals must also be included in the processor.

Requirement 2

The second requirement is that the segmented program be executable in a significantly smaller FL than the unsegmented program, without excessive use of other system resources. The two other system resources that are to be considered are: the additional central processing unit (CPU) time and input/output (I/O) time needed to load and unload the segments and the size of the file on which the absolute segments are stored.

For any given input program, there are a large number of correct ways to segment the program and still satisfy Requirement 1. One of these ways is to place all of the ROMs into one segment. Such a segmentation would not cause any large increases in the CPU or I/O time needed to execute the program and the absolute file would be small. However, this segmentation would not save any central memory. At the other end of the spectrum, a method may be found that will minimize the amount of memory required but causes an order of magnitude increase in execution time and creates an absolute file so large that it exceeds secondary storage space. Therefore, the processor must make an acceptable trade-off

between memory saved and the other additional resources needed.

Requirement 3

The third requirement is that all of the ROMs of the input program should be segmented. Since the input program may use ROMs from many different sources, the call graph should contain all of these ROMs and the relations between them. Some of these sources might be: several different files or a local, global, or system library.

Requirement 4

The fourth requirement is that the processor be easy to use. The reason for developing the processor is to reduce the programmer's work load. Any processor that does not do this would not be very useful. Therefore, the processor should be designed in such a manner that the user needs no knowledge of its internal workings. In other words, the processor should be as its name implies, automatic.

IV. Design

This chapter describes some of the concepts and algorithms that are used to implement SEGMENT. First, the input data which is necessary to completely define the input program is specified and an algorithm for obtaining this data is given. Next, algorithms are developed that determine the segmentation forest of the input program, the placement of the LCBs, and the placement of the ROMs that preset data in those LCBs. Then, several methods of modifying the segmentation forest to reduce loader activity are given. Finally, techniques for producing the loader directives are presented.

Input Data

The input data that is necessary to completely describe the program to be segmented are listed below:

(1) the names of the ROMs and their lengths,

(2) the name of the root (starting) ROM and the entry points in that ROM,

(3) which ROMs call which ROMs,

(4) the number of times each ROM calls every other ROM during execution of the program,

(5) the names of the LCBs and their lengths,

(6) which ROMs define which LCBs,

(7) which ROMs preset data in which LCBs,

(8) and which ROMs have indirect references to externals in which ROMs.

All of this data is stored in the loader tables of the ROMs of the program except the number of times each ROM calls another during execution. This piece of data cannot be known in advance of execution since the number of times a ROM is called may depend upon data read in by the program or on decisions made by the program during execution. Therefore, the assumption is made that all ROMs will be executed the same number of times.

The name of the root ROM and an entry point in that ROM will be supplied by the user. Also, since it is very difficult to detect indirect references, these too are supplied by the user. All of the remaining information can be organized, formatted, and written to a specified file by doing a "dummy" SEGLOAD of the program and requesting a load map of the segmented load. Load maps of basic loads do not list ROM references to LCBs so segmented loads will be used.

A segmented load map is made up of four records, samples of which are in Appendix C. The first record is a listing of all of the ROMs processed by the loader and the LCBs they define. Whatever purpose the empty second record serves is unknown to the author. The third record is a listing of all the errors, including data preset errors, that were discovered by the loader. A data preset error occurs when the loader detects that a ROM is trying to preset data in an LCB in another segment. The fourth record is a cross-reference map of which ROM calls which ROM.

Given a partial load sequence, the root ROM name, and

the name of an entry point, the following load sequence will produce a segmented load map. The partial load sequence can only contain LOAD, SLOAD, LIBLOAD, and LDSET control statements.

```
LDSET,MAP=SBX/MAP.
SEGLOAD.
partial load sequence
NOGO.
*eof
    TREE    root ROM name
    END     entry point name
```

This load map will not contain any data preset errors since all ROMs and LCBs are placed in the root segment by the loader.

To insure that all significant data preset errors are found, a second SEGLOAD is done with two segments instead of just one. The first segment contains the root ROM and all of the LCBs. The second segment contains all of the ROMs except the root ROM. Any ROM in the second segment that tries to preset data in an LCB in the first segment will cause an error during this SEGLOAD. These errors cause messages to be written to the third record of the load map. Each message includes the ROM and the LCB names involved. If the root ROM presets data in an LCB, then no error will be generated. This is not significant as LCBs defined by the root ROM will be placed in the root segment anyway.

This method of using two segments to force the preset errors causes the fourth record, the cross-reference data, to be invalid. Therefore, the records containing valid information from both of the segmented load maps are used as

the source of the input data.

Algorithm 1 describes how this is done. Note: the multiple SEGLOADs of Algorithm 1 may be implemented using a CYBER Control Language (CCL) procedure so that this algorithm can be made transparent to the user (Ref 8).

Algorithm 1

1. Use the root ROM name and the entry point name to generate directives that will place all ROMs and all LCBs in one segment.

2. Do a SEGLOAD with these directives and the partial load sequence.

3. Get the names of all of the ROMs and the LCBs from the first record of the load map produced by step 2.

4. Use these names to generate directives that place the root ROM and the LCBs in one segment and all remaining ROMs in a second segment.

5. Do a second SEGLOAD with these directives and the partial load sequence.

6. The first and third records from the load map of step 5 and the fourth record from the load map from step 2 now contain all the information necessary to describe the input program.

Depth-first Search

The method that is used in this paper to transform the call graph into a segmentation forest is called a depth-first search (DFS). This method of analyzing rooted graphs is described in several papers (Ref 5; Ref 6; Ref 7). The DFS is used here to identify break points in the call graph where the descendant ROMs may be separated from the graph to form a new level. The DFS is also used to identify the edges that complete cycles and redundant edges so they may be

deleted from the call graph.

The DFS insures that every edge between two reachable vertices of a rooted directed graph is traversed exactly once. The search begins by selecting a vertex v, the root ROM in this case, and placing v on a stack. Then, an arbitrary edge (v,w) is selected to visit vertex w. Now, let x be the most recently visited vertex, then the search is continued by selecting an untraversed edge (x,y) to visit vertex y. If y has already been visited, then another edge (x,z) is selected. If y has not been visited, then y is placed on the stack and the search continues from y. When all of the edges out of y have been traversed, the search backs-up to x by removing y from the stack and another untraversed edge out of x is selected. This continues until the search backs-up to the starting vertex v and there are no more untraversed edges out of v. At this point, the search is completed.

All of the traversed edges can be divided into two sets: those that led to a vertex that had not previously been visited and those that led to a vertex that had been visited before. Elements of the former set are called tree arcs. Elements of the latter are called back arcs. The tree arcs form a spanning tree of the reachable vertices in G. The back arcs may be further divided into three subsets based on the order in which the vertices that were visited by each back arc were stacked and unstacked.

At any time during the search, the stack will contain

all vertices that have been searched and still may have untraversed edges out of them. In addition, the order of the vertices in the stack will be a path from the starting vertex to the most recently visited vertex. If the vertices are numbered as they are placed on the stack and as they are removed from the stack, then the back arcs can be divided into the three following classes.

(1) The edges (v,w) where w is stacked when (v,w) is traversed are called fronds. A frond is the last edge in a cycle.

(2) The edges (v,w) where v is visited before w and w has been unstacked when (v,w) is traversed are called reverse fronds. A reverse frond is a redundant edge.

(3) The edges (v,w) where w is visited before v and w has been unstacked when (v,w) is traversed are called cross links. When (v,w) is a cross link, w is a lattice vertex.

Algorithm 2 is a recursive procedure for performing a DFS on a directed graph with a starting vertex s. It labels all reachable edges as tree arcs, fronds, reverse fronds, or cross links, and marks lattice vertices. In addition, the tree arcs define a father-son relation between all vertices in the graph.

The input graph G = (V,E) is stored as a set of adjacency lists A(v), for all v contained in V. Each vertice in G will have two attributes, NUMBER and SNUMBER. NUMBER(v)=m

means v was the mth vertice placed on the stack. SNUMBER(v)=n means v was the nth vertice removed from the stack. NUMBER(v)=0 if and only if v has not been visited. SNUMBER(v)=0 if and only if v has been visited but not backed-up from.

Algorithm 2

```
procedure CLASS(A,s)
    begin
        procedure SEARCH(v)
            begin
                m:=NUMBER(v):=m+1
                for all w in A(v) do
                    begin
                        if NUMBER(w)=0 then
                            begin
                                label (v,w) a tree arc
                                mark v as the father of
                                   w
                                SEARCH(w)
                            end
                            else if SNUMBER(w)=0 then
                                begin
                                    label (v,w) a frond
                                    delete (v,w)
                                end
                            else if NUMBER(v) lt NUMBER(w)
                               then
                                begin
                                    label (v,w) a reverse
                                       frond
                                    delete (v,w)
                                end
                            else
                                begin
                                    label (v,w) a cross
                                       link
                                    mark w as a lattice
                                       vertice
                                end
                        n:=SNUMBER(v):=n-1
                    end
            end
        m:=0
        n:=/V/+1
        for all   v   in V do NUMBER(v):=SNUMBER(v):=0
        SEARCH(s)
    end
```

Algorithm 2 identifies all of the break points by marking the lattice vertices. After the DFS has been accomplished, the lattice vertices and all of their descendants are moved to a new level. This new level is now searched to determine if there are any remaining break points.

Since it is probable that the subgraph in the new level will not have a root vertex, a pseudo root vertex is added to this subgraph and this pseudo root is connected to the subgraph by adding pseudo edges from the pseudo root to every vertex in the new level with an in-degree of zero.

Algorithm 3, which transforms a call graph into a Segmentation forest, is now presented.

<u>Algorithm 3</u>

```
procedure POSMOD(A,s)
    begin
        do until no lattice vertices are found by CLASS
            begin
                LEVEL:=0
                add a pseudo root to level LEVEL
                add pseudo edges to all vertices in
                   level LEVEL with an in-degree of
                   zero
                CLASS(A,s)
                for all vertices v in level LEVEL do
                    if v is marked as a lattice vertice
                       then
                        move v and all of its descen-
                           dants to the next level
                LEVEL:=LEVEL+1
            end
    end
```

Figs. 5 through 8 give an example of how Algorithm 3 would transform a call graph into a segmentation forest. Fig. 5 shows the call graph of a sample program with all of its vertices in one level. Fig. 6 is the same graph after

its edges have been labeled by Algorithm 2 and the vertices 5, 6, 7, 8, and 9 have been marked as lattice vertices (T for tree arc, F for frond, R for reverse frond, C for cross link, and L for lattice vertice). Fig. 7 shows the graph after it has been broken and the lattice vertices and their descendants have been moved to a new level. Level 0 has now been reduced to a tree; however, level 1 still is not a forest. In Fig. 8, Algorithm 2 has been applied a second time and the one lattice vertice, 9, has been moved to a third level. When Algorithm 2 is applied to level 2, no new lattice vertices are found so Algorithm 3 is terminated.

The DFS of a graph is not necessarily unique. The decision to visit vertex 4 before vertices 2 and 3 is completely arbitrary. Therefore, different searches of the same graph may produce different results. Fig. 9 is an example of another way to search Fig. 5. However, the edge labeling and lattice vertex marking within a particular graph will always be consistent. Different searches will produce different segmentation forests which may or may not affect the amount of central memory saved by the processor.

Labeled Common Blocks

After Algorithm 3, each segment contains one ROM. Now the LCBs are assigned to the NCA segment of the segments referencing them. This section gives algorithms for finding the NCA of one or more segments in the segmentation forest and for dealing with data preset errors. When

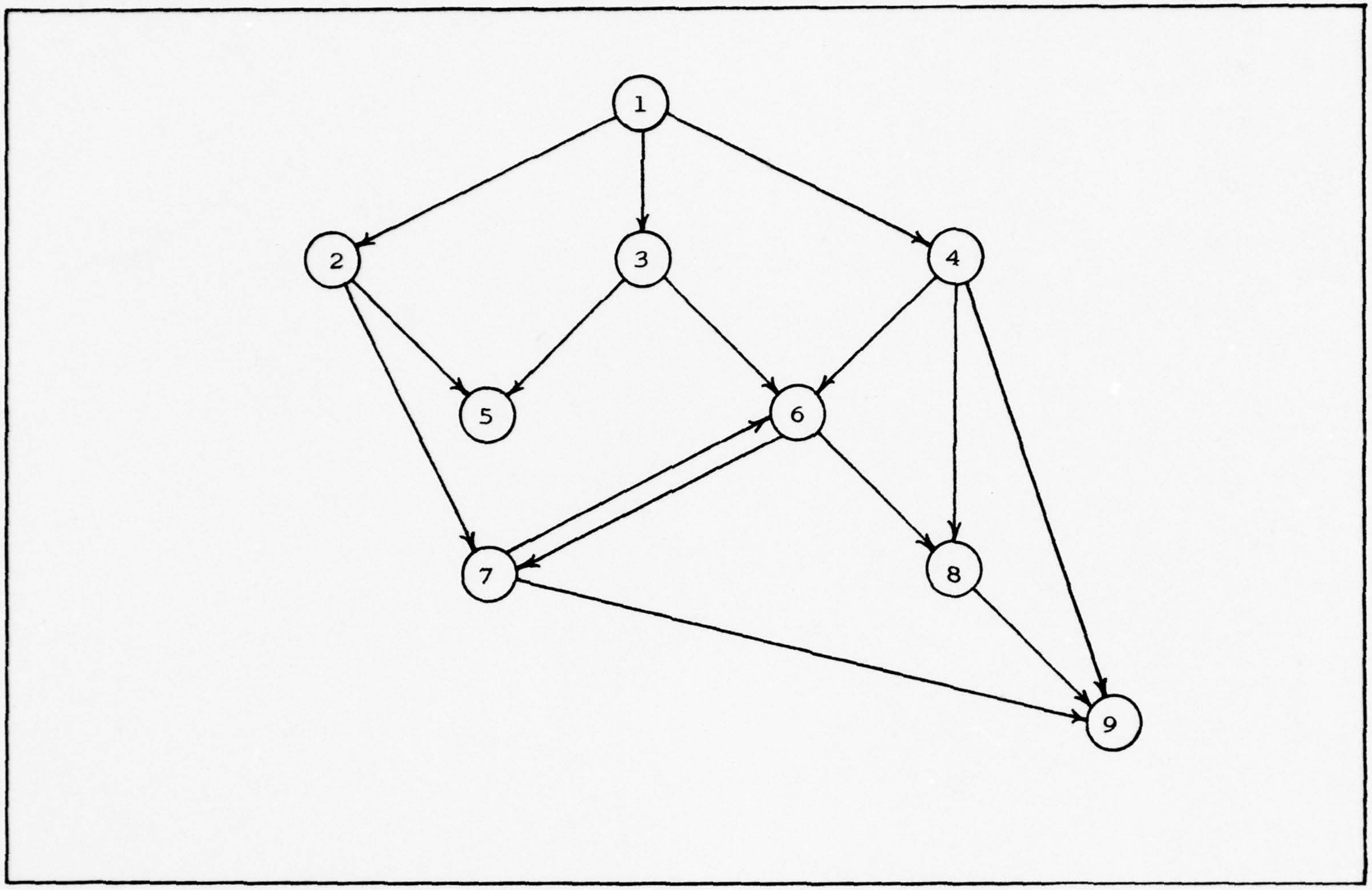


Fig. 5. A Sample Call Graph G

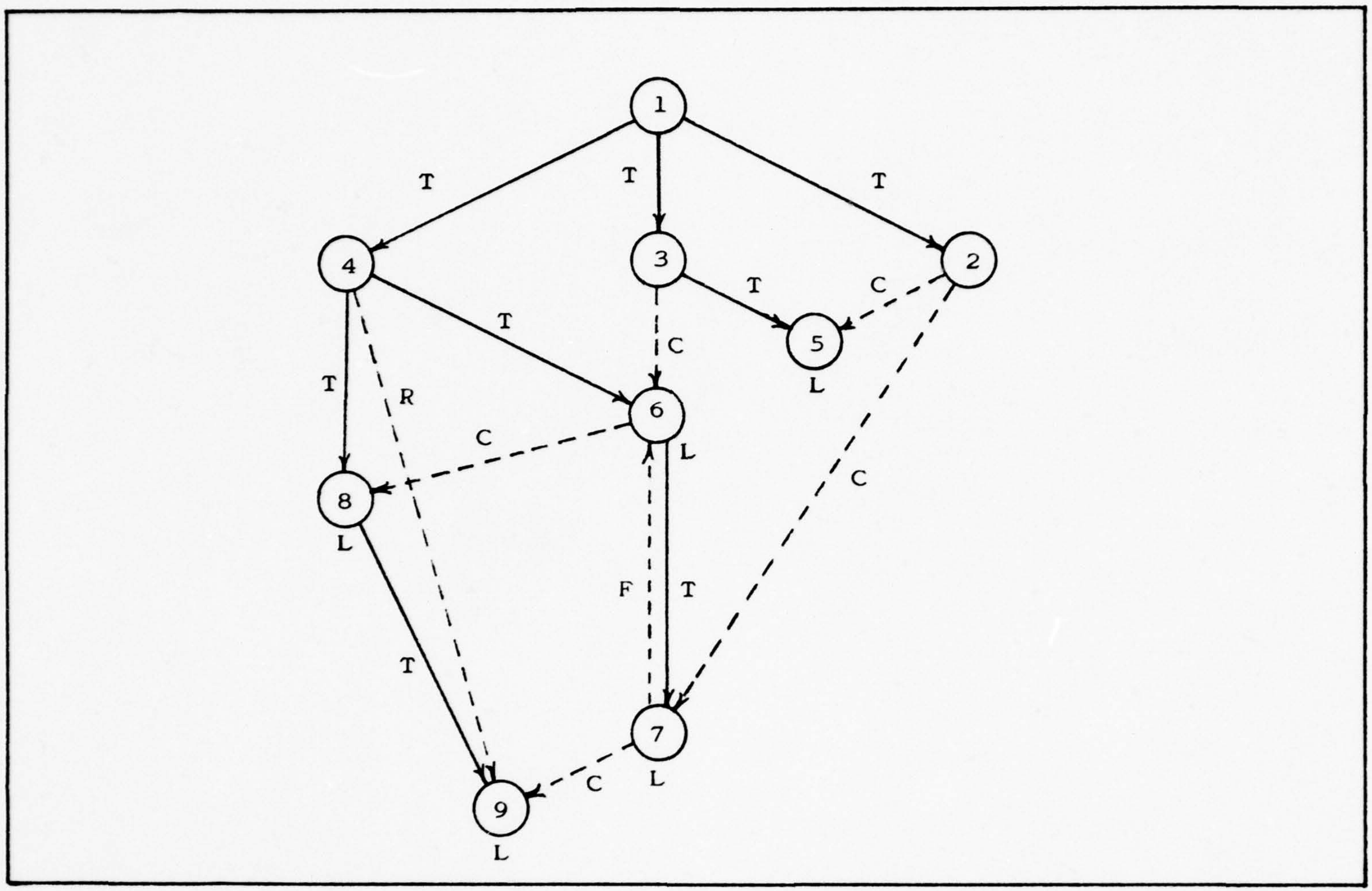


Fig. 6. A Depth-first Search of G

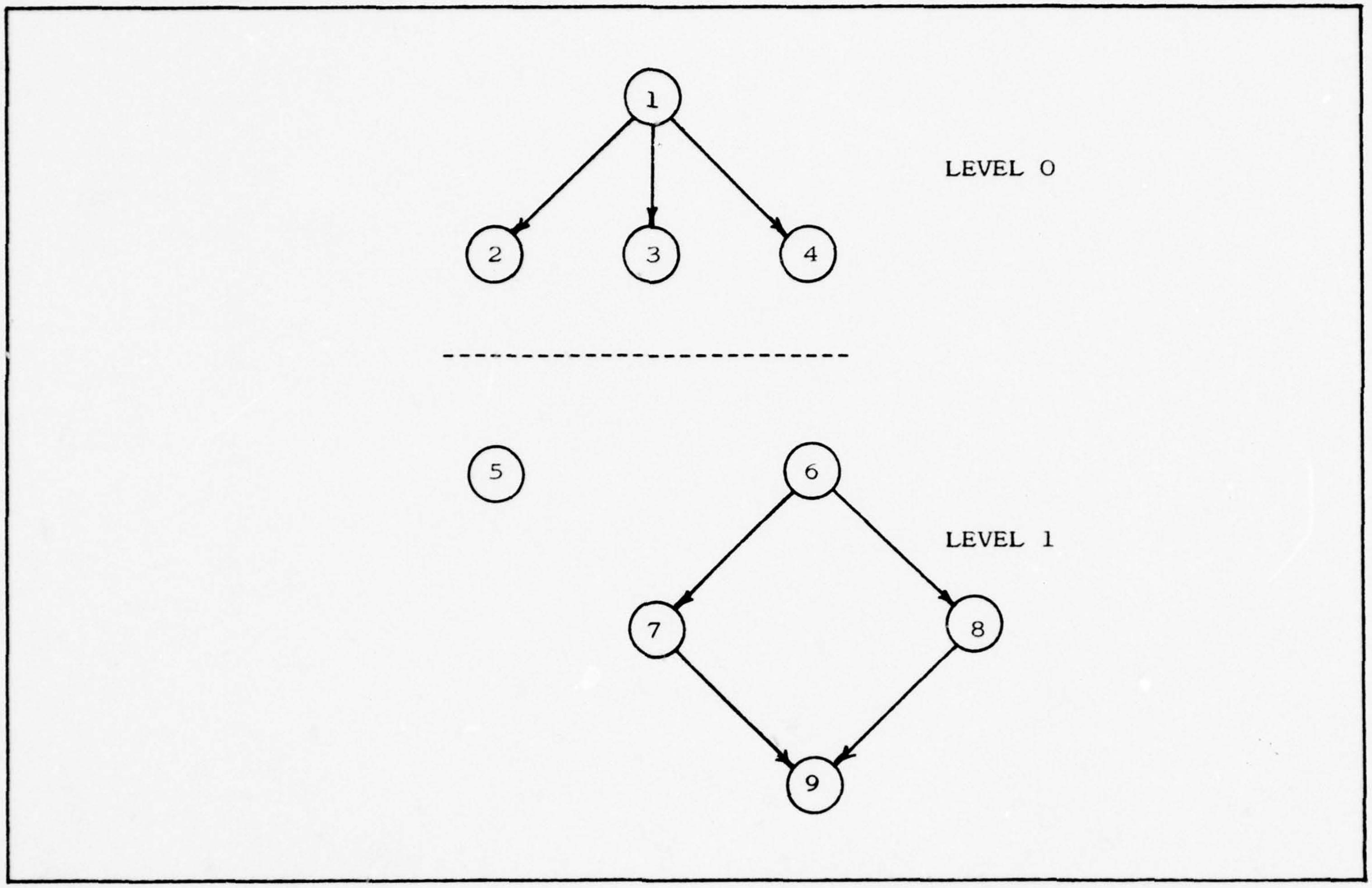


Fig. 7. The Graph G Split into Two Levels

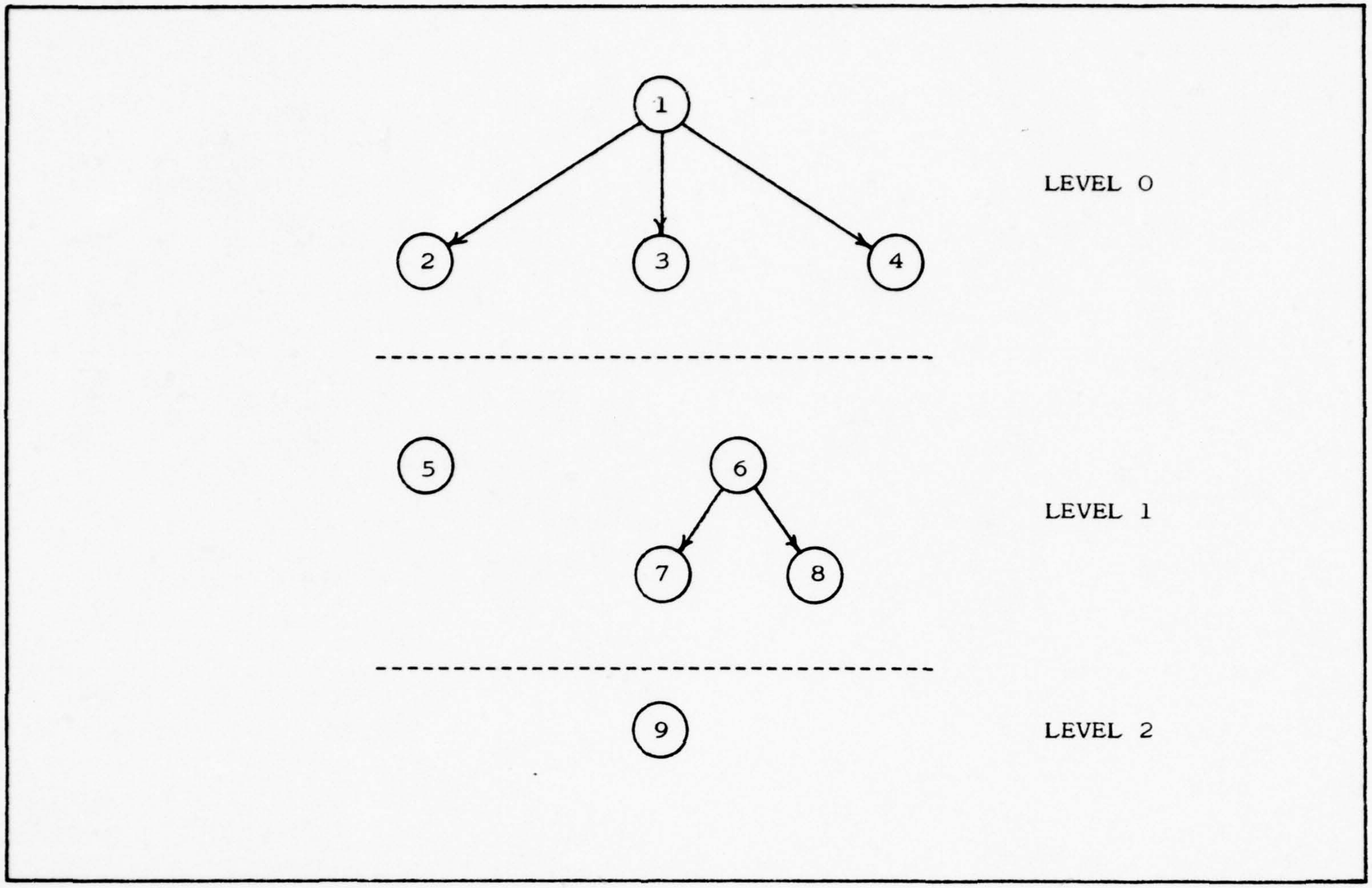


Fig. 8. Levels of Forests Produced by Algorithm 3

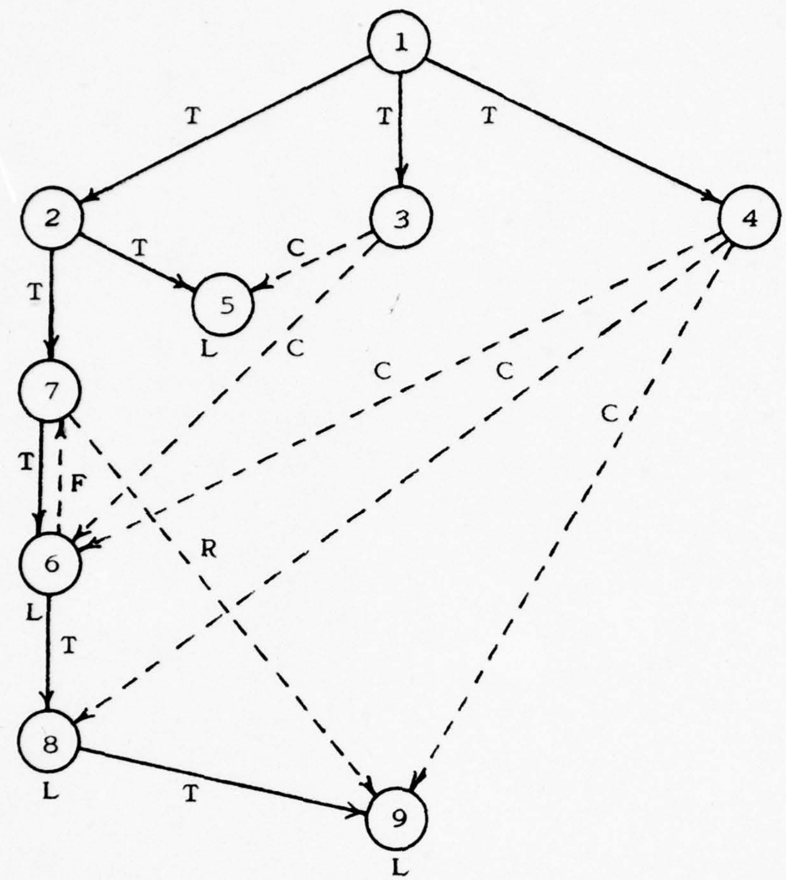


Fig. 9. A Second Depth-first Search of G

determining the NCA of a set of segments, it is assumed that all of the segments are in the same level of the segmentation forest. The LCBs that are referenced from more than one level must be placed in the root segment.

Algorithm 4 describes a function NEARER that returns the NCA of any two segments in a tree. NEARER uses the fact that every segment in a level was assigned a father segment by Algorithm 3. Two stacks of segments are created by following the fathers of each of the two input segments to the pseudo root of the level and placing each segment along the way on a stack. When the pseudo root is reached, the top of each stack is the pseudo root which is a common ancestor of the two input segments. If the top element of each stack is removed and the new top elements are still equal, then the segment represented by those elements is also a common ancestor of the two input segments. Therefore, the top elements of both stacks are removed until the new top elements are not equal. When this happens, the last element removed from the stacks is the NCA of the input segments.

<u>Algorithm 4</u>

```
integer procedure NEARER(v,w)
    begin
        push v and all of its ancestors onto VSTACK
        push w and all of its ancestors onto WSTACK
        while the top of VSTACK = the top of WSTACK do
            begin
                pop the top element off of VSTACK
                pop the top element off of WSTACK
            end
        NEARER := the last element popped off the stacks
    end
```

NEARER is now used by Algorithm 5 to position all LCBs

of a single level into their NCAs. The LCBs and the ROMs that reference them are stored in adjacency lists similar to the ROM adjacency lists. There is an adjacency list C(c) for each LCB.

Algorithm 5 first initializes the NCA of a particular c to the first element in C(c). The NCA is then updated as it is compared to all remaining elements of C(c).

<u>Algorithm 5</u>

```
procedure ANCSTR(C)
    begin
        for all levels i do
            begin
                for all c in C in i do
                    begin
                        NCA:=first element of C(c)
                        for all remaining elements
                          x of C(c) do
                            NCA:=NEARER (NCA,x)
                        if NCA is a pseudo root then
                            place c in the root segment
                        else place c in the NCA segment
                    end
            end
    end
```

ANCSTR and NEARER are now used to position all LCBs and to position ROMs that preset data in these LCBs. The first task that must be accomplished to do this is to place all LCBs referenced from more than one level into the root segment. Then, an iterative process is begun. First, all remaining LCBs are placed in their NCA. A list is then made for each ROM that presets data. Each list contains all the segments that a particular ROM tries to preset data in. Then, each ROM is moved to the NCA of the segments in its list. If no ROMs are moved during an iteration, then the loop is

stopped. Otherwise, the loop continues with LCBs being placed in their NCA. This iteration is necessary because a single ROM may preset data in several LCBs and a single LCB may be preset by several ROMs.

Algorithm 6 accomplishes the above tasks. It is assumed that a list showing which ROM preset data in which LCBs is available.

<u>Algorithm 6</u>

```
procedure POSCOM(C)
    begin
        for all c in C in more than one level do
            place c in the root segment
        CHANGE:=1
        while CHANGE = 1 do
            begin
                CHANGE:=0
                ANCSTR(C)
                for all ROMs v that preset data do
                    begin
                        make a list of segments contain-
                          ing LCBs preset by v
                        NCA:=first element of the list
                        for all other elements e in the
                          list do
                            NCA:=NEARER (NCA,e)
                        if NCA ne v then
                            begin
                                move v to segment NCA
                                CHANGE:=1
                            end
                    end
            end
    end
```

Algorithm 6 is the first algorithm to move a ROM to a new segment. Up to this point each ROM was placed in a segment of its own. When a ROM is moved, its segment is deleted and any ROMs or LCBs that were placed in that segment are moved with the original ROM. Also, the sons of the deleted segment must be made sons of the father of the deleted

segment. Algorithm 7 does these things.

Algorithm 7

```
procedure MOVEUP(SON,DAD)
    begin
        for all c in C do
            if c was placed in segment SON then
                place c in segment DAD
        for all v in V do
            begin
                if v was placed in segment SON then
                    place v in segment DAD
                if v is a son of SON then
                    make v a son of DAD
            end
        delete segment son
    end
```

Reductions

Originally, the call graph G of an input program had ROMs as vertices and calls between ROMs as edges. Now, after applying Algorithms 3 and 6, the segmentation forest G' is made of segments. Each vertex in G' is a segment of one or more ROMs and zero or more LCBs. Edges now define a father-son relationship between segments. Also, each edge in G' represents loading of a segment by SEGRES. Therefore, if the number of edges in G' can be reduced, then the loader activity during execution will be reduced. There are two reduction rules that may be applied to the segmentation forest that decrease loader activity but do not change the central memory required.

Rule 1. The first reduction rule is that all segments in G' with an in-degree of one and an out-degree of one may be combined into the father of that segment. This can be done quite easily. For any segment s in a tree, the fathers

of all of the segments in the tree are scanned to build a set of the sons of s. If this set contains only one element, then the conditions for Rule 1 are met and this only-son segment is moved into s using Algorithm 7.

Fig. 10a shows a segmentation tree; Fig. 10b is the same tree after Rule 1 has been applied. The number of edges is decreased but the central memory required for execution of the tree is the same.

Rule 2. Fig. 11 shows how the ROMs of Fig. 10 might be placed in central memory. The maximum amount of central memory that this program could require would be when segment 10 is loaded. This means that allowing segment 5 to overwrite segment 6 does not result in any memory savings.

Therefore, reduction Rule 2 states that two conflicting segments may be combined into one segment if this combination does not increase the amount of central memory necessary to execute the program. Rule 2 may be applied repeatedly to the same level of the segmentation forest and may result in segments, sub-trees, or trees of the segmentation forest being combined into a single segment. Fig. 12 shows how the tree of Fig. 10b might be modified by Rule 2.

Defining an optimal algorithm for Rule 2 is a very difficult task and no such algorithm was developed during this study. It should be obvious that such an algorithm would be very desirable to have as Fig. 10b contains seven edges whereas Fig. 12 contains only four edges.

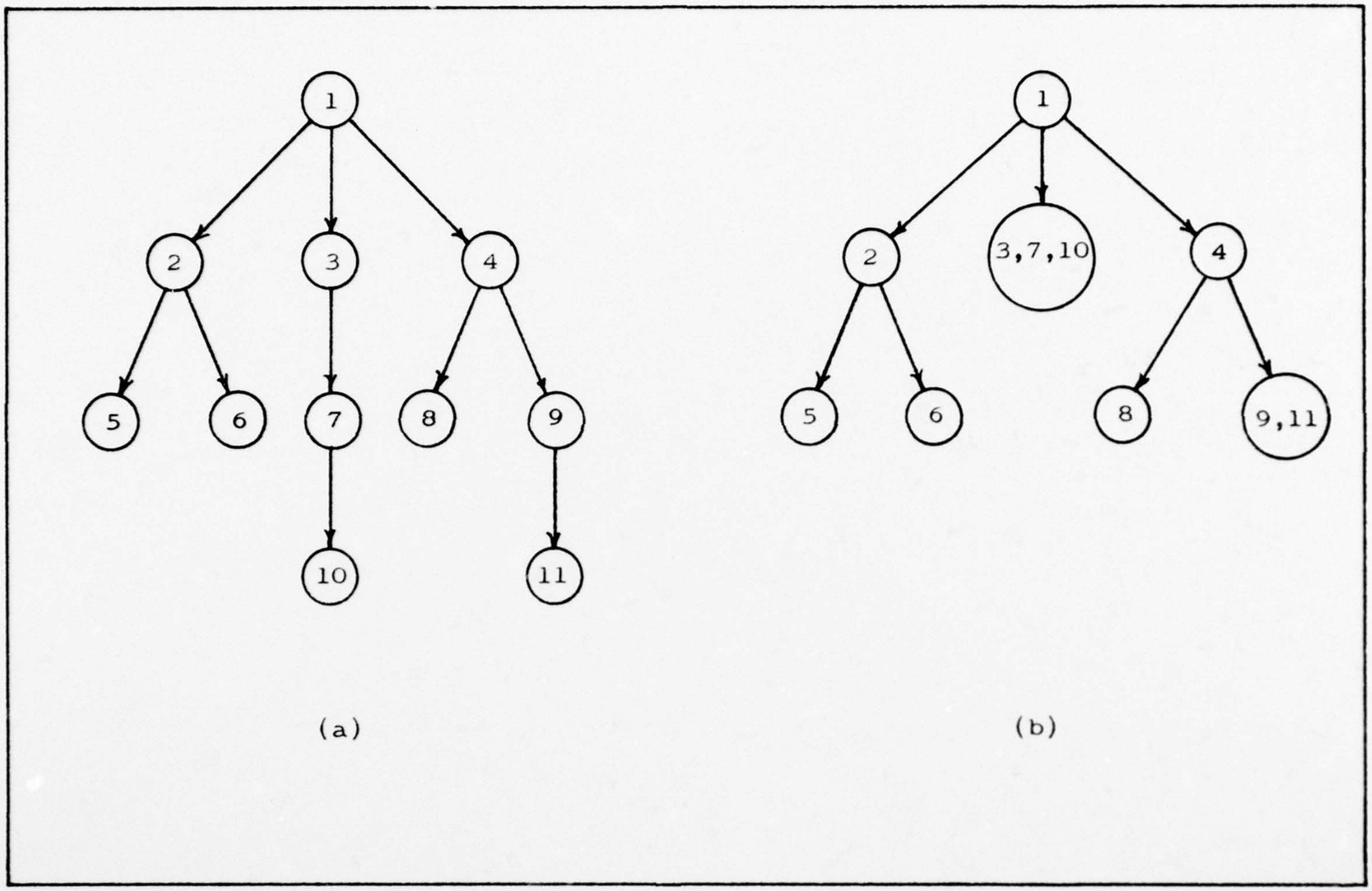


Fig. 10. Application of Reduction Rule 1 to a Segmentation Tree

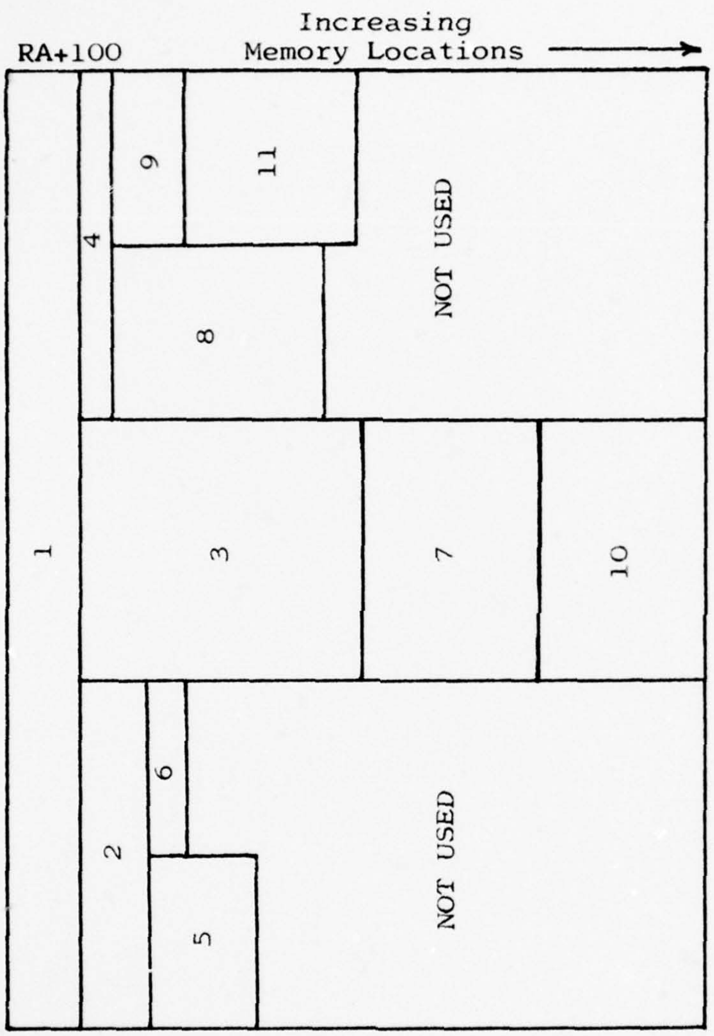


Fig. 11. Memory Map of Fig. 10

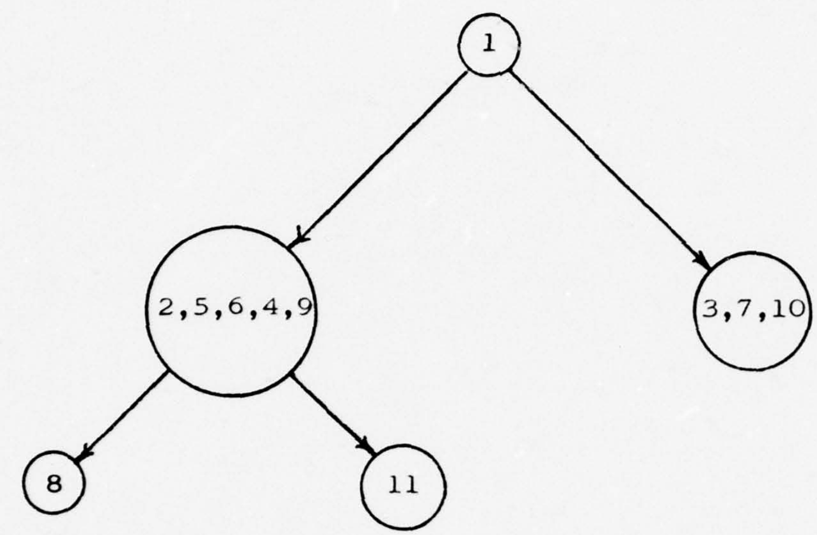


Fig. 12. Application of Reduction Rule 2 to Fig. 10

Generating Directives

The data which is required to generate the segmentation directives are:

(1) the names of all of the ROMs,

(2) the names of all of the LCBs,

(3) the ROMs and LCBs that are assigned to each segment,

(4) the segments that are assigned to each level of the segmentation forest,

(5) the father-son relations between the segments of each level,

(6) and an entry point name.

Even with these data, there are several ways of writing the segmentation directives, all of which will result in the correct execution of the input program.

The method of generating directives used in this thesis was chosen because it is a structured approach and it is easy to automate. Fig. 13 shows a segmentation forest with ROMs and LCBs assigned to the correct segments. The names of the ROMs and the LCBs that are assigned to each segment are listed inside each circle. An identifying segment number for each segment is shown just to the right of each circle. Each segment is given a name that is constructed by concatenating "S." with the level number of the segment (the level the segment is in plus one) and with the segment number of the segment. So the name given to segment 3 is S.01003 and the name given to segment 10 is S.02010. Each segment is also defined to be a tree with at least one vertex. The

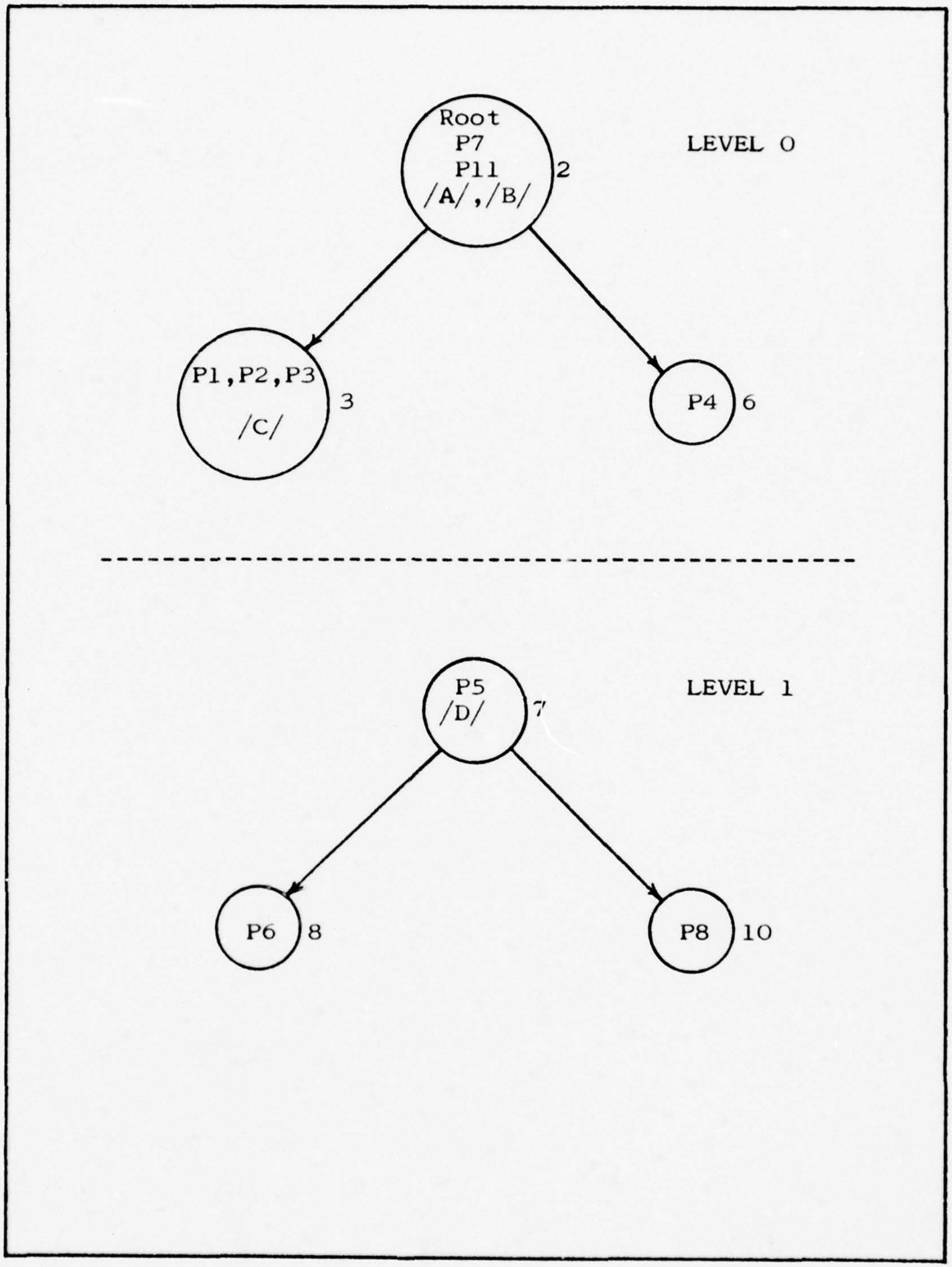


Fig. 13. A Sample Segmentation Forest with ROMs and LCBs Assigned

names of the trees are constructed in the same manner as the segment names except that the "S." is changed to "T.". Some other conventions used are: only one ROM or LCB is placed in a segment with a single directive, all LCBs are saved, and all of the directives for a particular level are generated together. The TREE directives can be arbitrarily long, based on the out-degree of a particular segment, and they may have to be continued on one or more continuation cards.

The directives that would be generated for the program of Fig. 13 are listed below.

```
S.01002  GLOBAL   A-SAVE
S.01002  GLOBAL   B-SAVE
S.01003  GLOBAL   C-SAVE
S.01002  INCLUDE  ROOT
S.01002  INCLUDE  P7
S.01002  INCLUDE  P11
S.01003  INCLUDE  P1
S.01003  INCLUDE  P2
S.01003  INCLUDE  P3
S.01006  INCLUDE  P4
T.01002  TREE     S.01002-(T.01003,T.01006)
T.01003  TREE     S.01003
T.01006  TREE     S.01006
         LEVEL    1
S.02007  GLOBAL   D-SAVE
S.02007  INCLUDE  P5
S.02008  INCLUDE  P6
S.02010  INCLUDE  P8
T.02007  TREE     S.02007-(T.02008,T.02010)
T.02008  TREE     S.02008
T.02010  TREE     S.02010
         LEVEL    2
         END      SAMPLE
```

V. Segment Module Specifications and Interfaces

In the last chapter, the algorithms necessary to segment a user program were described. This chapter assigns these algorithms to a set of modules and defines the interfaces between them.

SEGMENT

The processor that was developed for this thesis is a CYBER Control Language (CCL) procedure called SEGMENT. This procedure communicates with the user through the files shown in Fig. 14. The user supplies five files to SEGMENT, two input files and three output files. All five files have default names if the user does not specify one or more file names in his call to SEGMENT. The two input files are a set of directives to SEGMENT and a partial load sequence which insures that the proper ROMs are loaded.

The directives provide SEGMENT with three pieces of information: (1) the name of the root ROM, (2) the name of an entry point in the root ROM, and (3) a method of building the call graph from the ROMs.

The call graph is made out of ROMs from different files and libraries which are specified in the partial load sequence. There are four methods of building the call graph out of these ROMs:

(1) All ROMs are combined into a single call graph.

This is the normal method of building the call graph

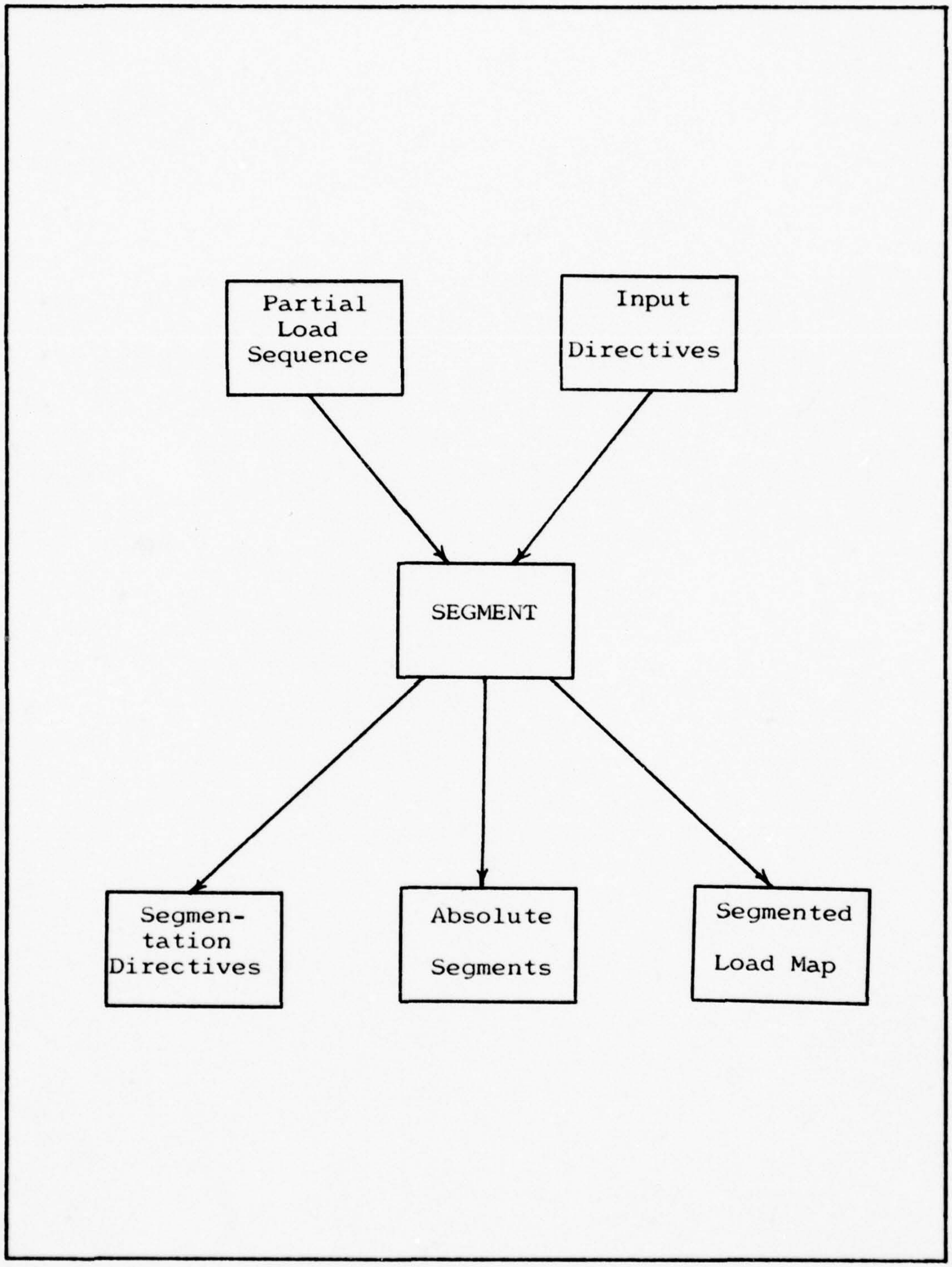


Fig. 14. User - SEGMENT Interfaces

if there are no indirect references to externals or any other peculiarity in the ROMs that prohibit putting any two ROMs into different segments.

(2) All ROMs, except those from files or libraries that are specified to be deleted, are combined into a single call graph. This is the usual way of handling indirect references.

(3) The ROMs of each file and library are combined into a separate call graph for each file or library. Since the data base used by SEGMENT is of fixed size, using this method reduces the number of edges in the call graph and may allow a larger program to be stored.

(4) All ROMs, except those from specified files or libraries, are combined into separate call graphs as in the preceding method. For some programs, deleting one or more files or libraries using method 2 above may result in the single call graph becoming disjoint. This method creates separate call graphs which can be handled by SEGMENT.

The format for the input directives to SEGMENT is given in Appendix B. The default file name for the input directives is INPUT.

The partial load sequence is a series of LOAD, SLOAD, LIBLOAD, and LDSET control statements which will be used by SEGMENT to load the proper ROMs with the correct loader options (Ref 1). The default file name for the partial load

sequence is INPUT.

The three output files produced by SEGMENT are the segmentation directives, with a default file name of PUNCH; the file of absolute segments, with a default file name of ABS; and the load map of the segmented load which produced ABS, with a default file name of OUTPUT. There is a currently sixth file that is not shown on Fig. 14 which is used to bring out diagnostic information about how the directives were generated. The default name of this file is OUTPUT.

The function of SEGMENT is to implement Algorithm 1 and to supply its descendants, shown in Fig. 15, with the proper files. Fig. 16 is a flow diagram of how SEGMENT does this using the four FORTRAN programs SEGO, SEG1, SEG2, and SEG3 and the segmenting loader. Each module in Fig. 16 inputs data from one or more files and writes data to one or more output files. The five files that are used to communicate with the user are marked with asterisks.

SEGO

The function of SEGO is to produce a CCL procedure containing a load sequence for a segmented load given a partial load sequence that describes which ROMs to load and how to load them. This CCL procedure is then called by SEGMENT to perform each of the three SEGLOADs shown on Fig. 16. SEGO interfaces SEGMENT through two files, the input file containing the partial load sequence and the output file containing the CCL procedure.

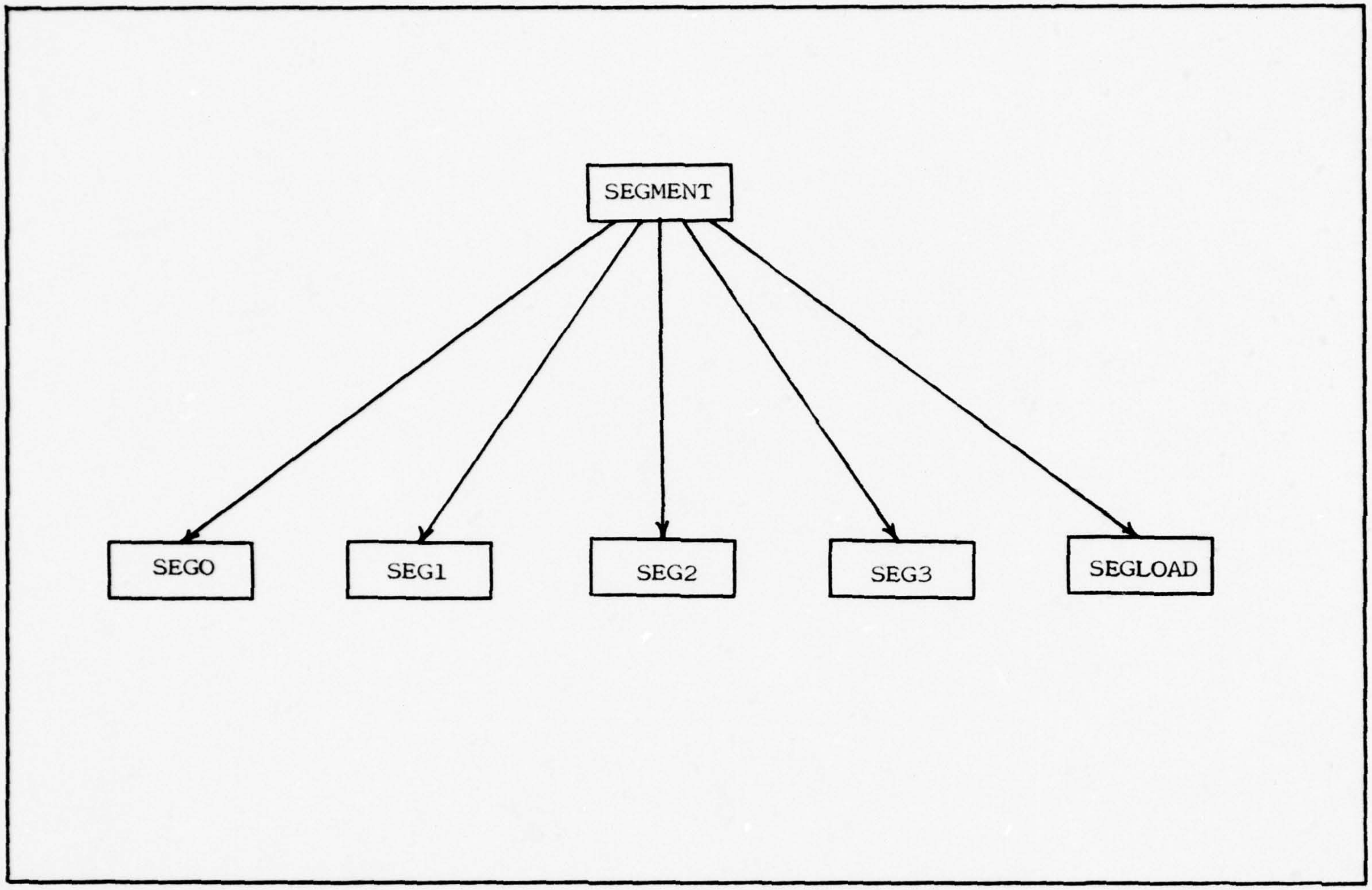


Fig. 15. Module Breakdown of SEGMENT

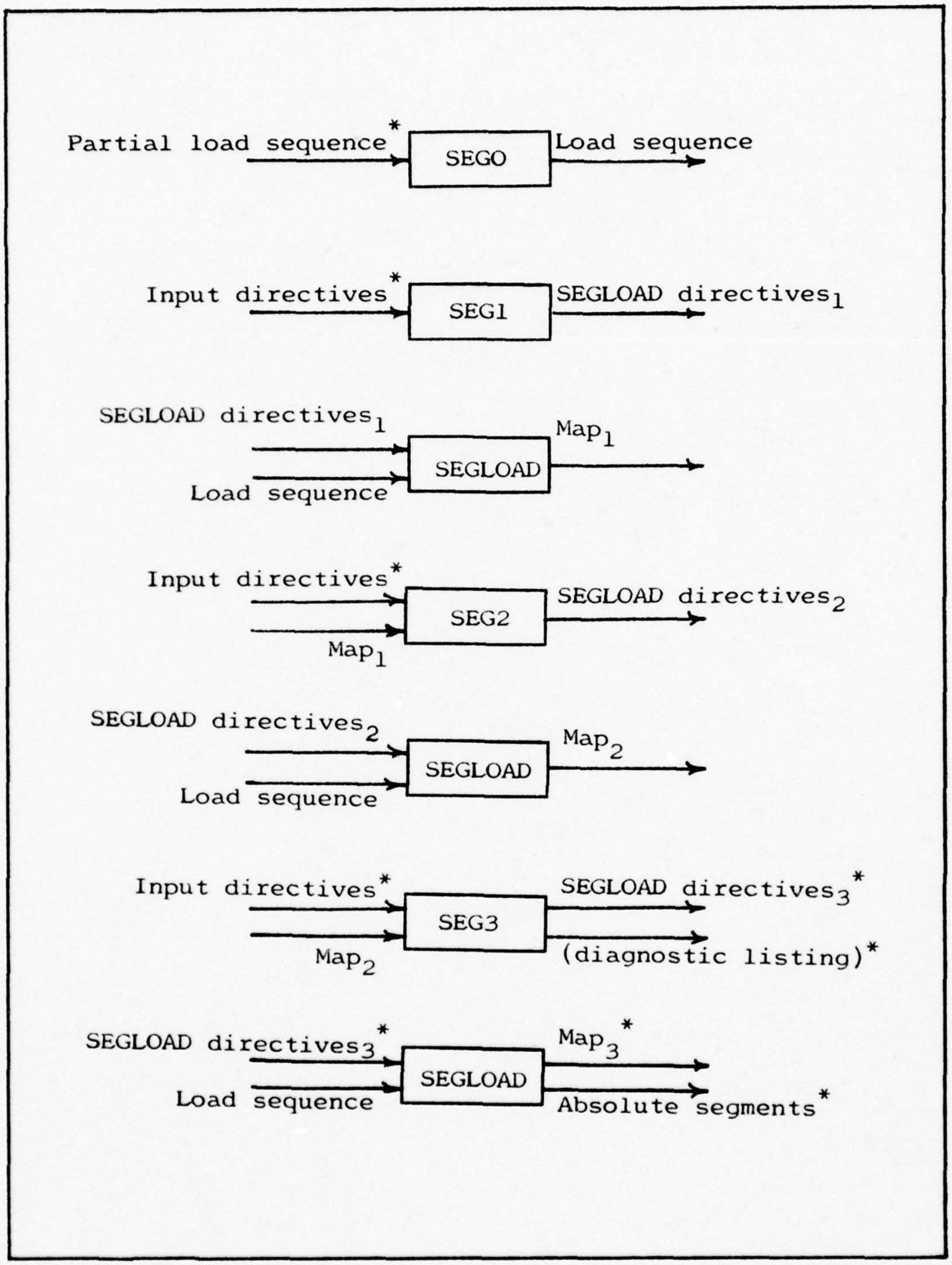


Fig. 16. A Flow Diagram of SEGMENT

SEG1

The function of SEG1 is to implement step 1 of Algorithm 1, i.e., it generates two Segmentation directives that place all ROMs and LCBs in the root segment. SEG1 interfaces SEGMENT through two files: the input directives file and an output file containing the two segmentation directives.

SEG2

The function of SEG2 is to implement steps 3 and 4 of Algorithm 1, i.e., it produces directives that force all data preset errors to be included in a subsequent segmented load map. SEG2 interfaces with SEGMENT through three files: the input directive file, the first record of the segmented load map produced with the directives from SEG1, and an output file containing the required segmentation directives.

SEG3

The function of SEG3 is to produce the Segmentation directives that will be output to the user. SEG3 interfaces with SEGMENT through five files. The first three files are the three segmented load map records described in step 6 of Algorithm 1. The fourth file is the input directives file. The fifth file is an output file that contains the segmentation directives that are passed to the user. SEG3 also produces the diagnostic file.

SEG3 is made up of 35 FORTRAN modules and a global data structure of 30 LCBs. Fig. 17 depicts the call graph of the

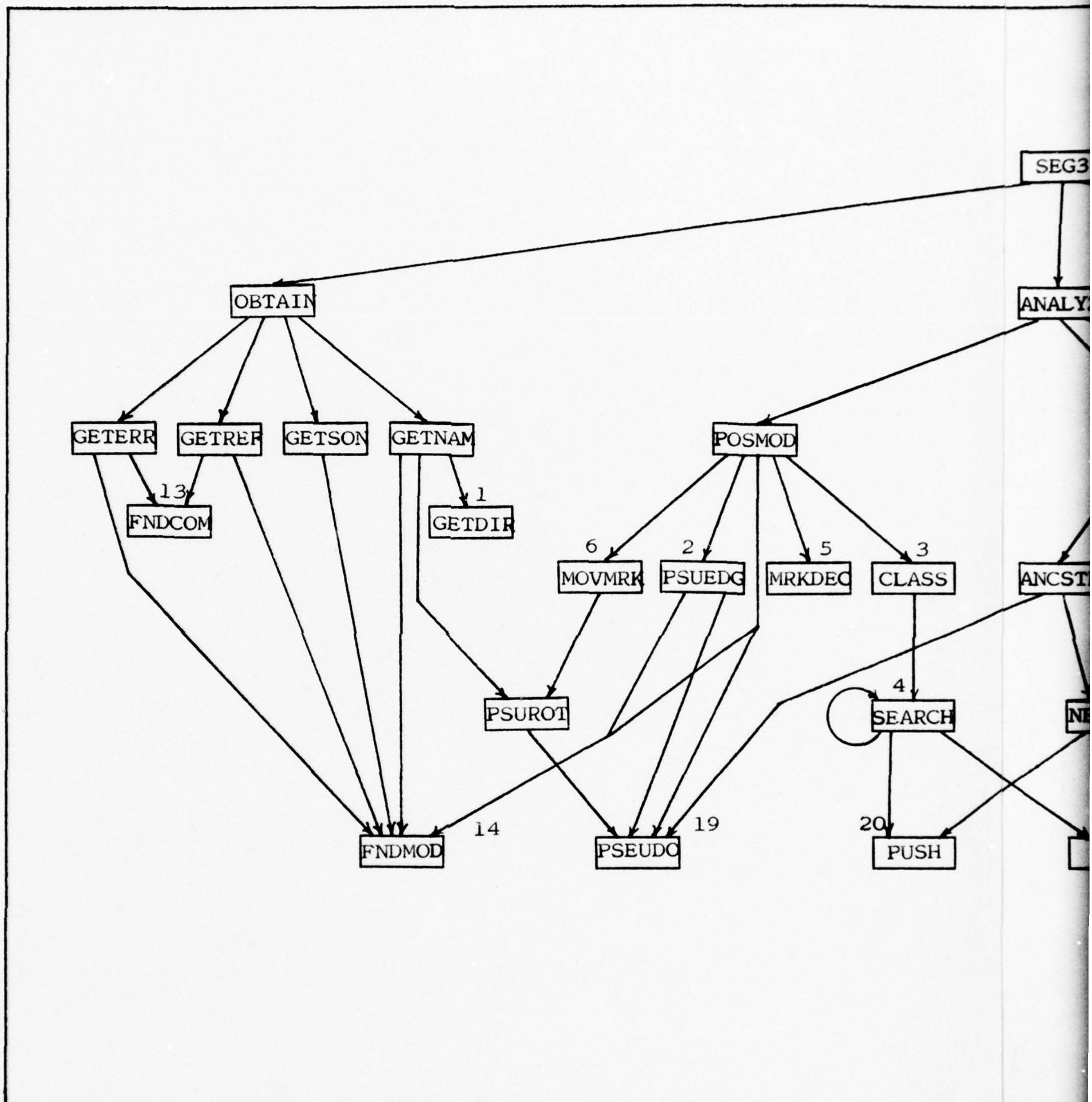
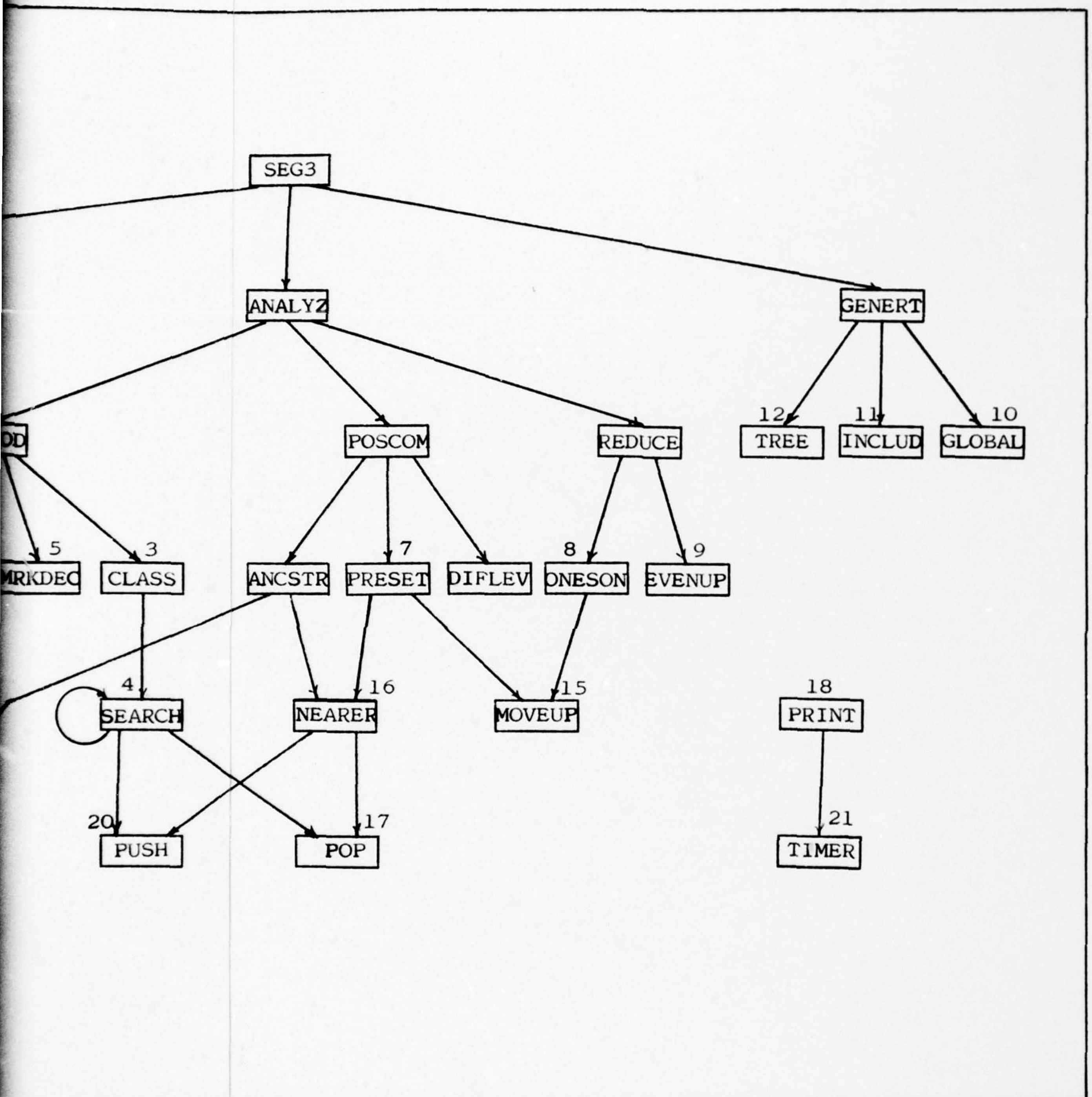


Fig. 17. Module Call Graph of SEG3



SEG3
ANALYZ
GENERT
DD
POSCOM
REDUCE
12 TREE
11 INCLUD
10 GLOBAL
5 MRKDEC
3 CLASS
ANCSTR
7 PRESET
DIFLEV
8 ONESON
9 EVENUP
4 SEARCH
16 NEARER
15 MOVEUP
18 PRINT
20 PUSH
17 POP
21 TIMER

2

SEG3 modules with keys into Table II. Table II gives a listing of all module-to-module interfaces between the modules of SEG3. All module-to-module interfaces are done using formal parameters on the modules. Any module not listed in Table II does not use any formal parameters.

In order for a module to interface the data base, the module must reference the LCB holding the required piece of data. Each element of the data base is a separate LCB. Table III gives a complete listing of all of the module-to-data base interfaces. On Table III, an O (for output) means that the module changes the value of the LCB in the data base but does not use the original value for any calculations. An I (for input) means that the module uses the value of the LCB but does not change it. A B (for both) means that the module both uses the value for some calculation and also changes that value.

<u>SEG3 Data Base</u>

The SEG3 data base consists of 30 LCBs with one data base element, either a simple variable or a singly subscripted array, per LCB. This method of communicating between the modules and the data base was chosen for several reasons. First, it greatly reduces the number of parameters in each subroutine call. Second, it limits access to the data base only to those modules that reference the data's LCB. If parameters or a single large LCB were used, then nearly all the modules would have access to all the information in the data base. Third, this method makes it much

Table II
Module-to-module Interfaces Keys from Fig. 17

KEY	CALLED MODULE	IN	OUT
1	GETDIR	- - - -	START,OPTION,BADLIB, NBADLIB
2	PSUEDG	LEVEL	- - - -
3	CLASS	S	- - - -
4	SEARCH	V,RSEARCH,M,N,NUMBER, SNUMBER	M,N,NUMBER,SNUMBER
5	MRKDEC	- - - -	CHANGE
6	MOVMRK	LEVEL	- - - -
7	PRESET	- - - -	CHANGE
8	ONESON	LEVEL	- - - -
9	EVENUP	LEVEL	- - - -
10	GLOBAL	LEVEL	- - - -
11	INCLUD	LEVEL	- - - -
12	TREE	LEVEL	- - - -
13	FNDCOM	NAME	FNDCOM
14	FNDMOD	NAME	FNDMOD
15	MOVEUP	SON,DAD	- - - -
16	NEARER	V,W	NEARER
17	POP	STACK,NSTACK	STACK,NSTACK,VAR
18	PRINT	TITLE	- - - -
19	PSEUDO	LEVEL	PSEUDO
20	PUSH	STACK,NSTACK,MSTACK, VAR	STACK,NSTACK
21	TIMER	TITLE	- - - -

Table III
SEG3 Module to Data Base Interfaces

	Group 1		Group 2										Group 3				Group 4						Group 5				Group 6			
	STARTNO	ENTRYPT	NOLEVEL	MAXMOD	NOMOD	MODULE	MODLEN	MODLINK	MODLEV	FATHER	MODSEG	MOVMOD	MAXSON	NOSON	SONMOD	SONLINK	MAXCOM	NOCOM	COMMON	COMLIN	COMLINK	COMSEG	MAXREF	NOREF	REFMOD	REFCOM	MAXPRE	NOPRE	PREMOD	PRECOM
SEG3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OBTAIN	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GETNAM	O	-	O	O	O	O	O	O	O	O	O	O	-	-	-	-	O	O	O	O	O	O	-	-	-	-	-	-	-	-
GETDIR	-	O	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GETREF	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	O	-	O	O	O	O	-	-	-	-
GETSON	I	-	-	-	-	-	-	B	I	-	-	-	O	O	O	O	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GETERR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	O	O	O	O
ANALYZ	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
POSMOD	-	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PSUEDG	I	-	-	-	I	-	-	B	I	-	-	-	I	B	O	B	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CLASS	-	-	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SEARCH	-	-	-	I	-	-	-	I	-	O	-	O	-	-	B	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
MRKDEC	-	-	-	-	I	-	-	I	-	-	-	B	-	-	I	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
MOVMRK	-	-	I	-	I	-	-	I	B	O	-	B	-	-	B	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
POSCOM	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DIFLEV	I	-	-	-	-	-	-	-	I	-	-	-	-	-	-	-	-	I	-	-	I	O	-	-	I	I	-	-	-	-
ANCSTR	I	-	-	-	-	I	-	-	-	-	-	-	-	-	-	-	-	I	-	-	I	B	-	-	I	I	-	-	-	-

Table III--Continued

	Group 1		Group 2										Group 3				Group 4						Group 5				Group 6			
	STARTNO	ENTRYPT	NOLEVEL	MAXMOD	NOMOD	MODULE	MODLEN	MODLINK	MODLEV	FATHER	MODSEG	MOVMOD	MAXSON	NOSON	SONMOD	SONLINK	MAXCOM	NOCOM	COMMON	COMLIN	COMLINK	COMSEG	MAXREF	NCREF	REFMOD	REFCOM	MAXPRE	NOPRE	PREMOD	PRECOM
PRESET	-	-	-	-	-	-	-	-	-	-	O	-	-	-	-	-	-	-	-	-	-	O	-	-	-	-	-	I	I	I
REDUCE	-	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ONESON	-	-	-	-	I	-	-	-	I	B	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
EVENUP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GENERT	-	I	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GLOBAL	-	-	-	-	-	-	-	-	I	-	-	-	-	-	-	-	-	I	I	-	-	I	-	-	-	-	-	-	-	-
INCLUD	-	-	-	-	I	I	-	-	I	-	I	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
TREE	-	-	-	-	I	-	-	-	I	I	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FNDCOM	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	I	I	-	-	-	-	-	-	-	-	-	-	-
FNDMOD	-	-	-	-	I	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
MOVEUP	-	-	-	-	I	-	-	-	-	B	B	-	-	-	-	-	-	I	-	-	-	B	-	-	-	-	-	-	-	-
NEARER	-	-	-	-	-	-	-	-	-	I	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
POP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PRINT	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
PSEUDO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PSUROT	-	-	B	I	B	O	O	O	O	O	O	O	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PUSH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
TIMER	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

easier to understand the module-to-data base interfaces and, hence, makes it easier to change and debug the system. Finally, this technique is suggested as a good way of structuring data for segmentation. SEGMENT will be used to segment SEG3 to verify this.

The names of the LCBs and the variable names in each LCB are identical to simplify program understanding. For example, the variable ENTRYPT is the only variable in LCB /ENTRYPT/ which is declared as COMMON/ENTRYPT/ENTRYPT in all modules requiring access to the variable ENTRYPT.

The SEG3 data base is divided into six groups. Group 1 contains the entry point name and an index for the root ROM. Group 2 contains the names of all of the ROMs and information about each ROM. Group 3 contains the adjacency lists for each of the ROMs in group 3. These lists define the edges of the call graph. Group 4 contains the names of all of the LCBs and information about each LCB. Group 5 contains the adjacency list of the ROM that reference the LCBs in group 4. Group 6 contains the data preset information. Each of these groups will now be discussed in detail.

Group 1 consists of two simple variables in two LCBs. ENTRYPT is the entry point name of the program being segmented and is obtained from the input directives. STARTNO is the index of the user defined root ROM which is stored in group 2.

Group 2 consists of ten LCBs. Seven of these are singly subscripted arrays that can be thought of as a single 7*n

array where n is the number of ROMs stored in group 2. The seven arrays are named: MODULE, MODLEN, MODLINK, MODLEV, FATHER, MODSEG, and MOVMOD. The ith element of all seven arrays contain information about the same ROM.

A description of the ith element of each of these arrays is given below:

MODULE(i) is the name of the ith ROM in the program being segmented.

MODLEN(i) is the word length of the ith ROM.

MODLINK(i) is a pointer to the start of the ROM adjacency list of the ith ROM.

MODLEV(i) is the segmentation level of the ith ROM.

FATHER(i) is the immediate ancestor of ROM i in the segmentation forest and is assigned by Algorithm 2.

MODSEG(i) is the segment that ROM i will be placed into. Initially, MODSEG(i) will equal i, but this will change when the presetting ROMs are moved and the reduction rules are applied.

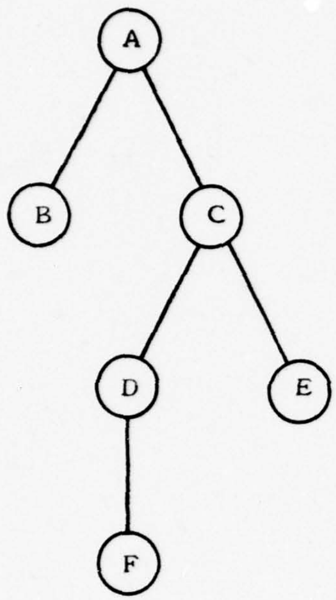
MOVMOD(i) is a flag which indicates whether or not ROM i is part of the spanning tree of the level currently being searched. If MOVMOD(i) is less than or equal to three, then i is not a lattice vertex. If MOVMOD(i) is greater than or equal to four, then i is either a lattice vertex or a descendant of a lattice vertex and must be moved to a new level.

Three simple variables are also part of group 2. NOMOD is the number of ROMs stored in the data base. MAXMOD is the maximum number of ROMs that the data base can hold and the number of segmentation levels that divide the ROMs is maintained in NOLEVEL.

Group 3 consists of four LCBs. NOSON and MAXSON are the number of entries in all of the ROM adjacency lists and the maximum number of entries allowed, respectively.

The function of the two arrays in group 3, called SONMOD and SONLINK, can best be shown by a diagram that depicts how groups 2 and 3 are linked together. Fig. 18 shows part of a call graph after the DFS has been performed and the fathers of all the ROMs have been assigned. ROM A has no father since it is the root. ROM A does have descendants though, and MODLINK(1) points to the adjacency list which contains A's sons. Therefore, the first son of ROM A is located at SONMOD(3) which is a 3. This means that A's first son is MODULE(3) or C. SONLINK(3) says that A has a second son at SONMOD(4) which is B. SONLINK(4) is 0 so A has only two sons.

The ordered pair (MODULE(1),SONMOD(3)) = (1,3) represents an edge in the call graph from A to C. When ROMs are moved to a new level, the edges between the levels are deleted by setting the corresponding SONMOD to 0. For example, if ROM C and its descendants are moved to a new level, then the edge (1,3) is deleted by setting SONMOD(3) to 0. The edge from A to B still exists since SONLINK(3) still



(a)

MODULE	MODLINK	FATHER	INDEX	SONMOD	SONLINK
A	3	0	1	4	5
B	0	1	2	6	0
C	1	1	3	3	4
D	2	3	4	2	0
E	0	3	5	5	0
F	0	4	6		

(b) (c)

Fig. 18. The Linked Lists Representing a Call Tree

points to SONMOD(4).

The order in which data is stored in these lists is dependent on the load map. This order determines the order in which the DFS will be performed. It should also be noted that the adjacency lists and the FATHERs list link the ROMs together in both directions. The adjacency lists define the call graph and they are used to perform the DFS. The FATHERs list is produced by the DFS. This list defines the segmentation forest and is used to produce the segmentation TREE directives.

Group 4 consists of six LCBs. NOCOM and MAXCOM are the current and maximum numbers of LCBs in the data base. The four remaining variables are arrays that may be thought of as a single 4*m array where m is the number of LCBs in the program being segmented. The four singly subscripted arrays are named: COMMON CONLEN, CONLINK, and COMSEG. The ith element of each of these arrays correspond to one another.

The functions of the ith element of each array is listed below:

COMMON(i) is the name of the ith LCB in the program being segmented.

COMLEN(i) is the word length of LCB i.

COMLINK(i) is a pointer to the start of the reference adjacency list for LCB i.

COMSEG(i) is the segment where LCB i is placed.

Group 5 consists of four LCBs that serve the same function as those in group 3 except that group 5 is the adjacency

lists for the stored LCBs in group 4. NOREF and MAXREF refer to the length of adjacency list. REFMOD(i) is the index of the ROM in group 2 that references an LCB in group 4. REFLINK is analogous to SONLINK in group 3. Groups 4 and 5 are linked together in the same manner as groups 2 and 3 are in Fig. 18.

Group 6 consists of four LCBs. There are two arrays, PREMOD and PRECOM, that hold all significant information concerning data preset errors. This data is stored such that it can be said that MODULE(PREMOD(i)) presets data in COMMON(PRECOM(I)). The simple variables NOPRE and MAXPRE are the current and maximum numbers of data preset errors stored in group 6.

SEG3 Modules

The function of each module in SEG3 is given in this section. Also, any general comments about the module implementation and module-to-module interfaces will also be given.

OBTAIN. This module is responsible for building the data base from information on the four input files to SEG3. The position of all the desired information on the load map is known and constant (see Appendix C), so the tab (T) format specification is used to read in all data from the load map (Ref 9:I-10-34). This eliminates the need for doing any pattern matching.

GETNAM. GETNAM takes all of the ROM and LCB names and lengths off of the first record of the load map and places them in groups 2 and 4. When a ROM or an LCB is added, all

of the appropriate variables in that group are updated or initialized.

GETNAM places the ROMs in the data base in one of the four ways described earlier. After all ROMs have been read in, the index of the root ROM is found and placed in STARTNO.

GETNAM will cause an abnormal termination of SEG3 if any of the following four errors are detected: (1) if an early end of file is encountered on the load map, (2) if there are too many ROMs, (3) if there are too many LCBs, or (4) if the root ROM name cannot be found in the data base.

GETDIR. GETDIR returns four variables to GETNAM: START, OPTION, BADLIB, and NBADLIB. START is the root ROM name; OPTION is one of two methods for positioning the ROMs; BADLIB is an array with up to five "bad" file or library names; and NBADLIB is the number of names in BADLIB. This data is read off of the input directives file.

The only information that must be on this file is the root ROM name. If this is not found, then SEG3 is terminated. If no entry point name is found, then it is assumed to be the same as the root ROM name. OPTION defaults to a value that places all of the ROMs in a single level. All libraries are processed if no "bad" file or library names are provided.

GETREF. This routine is responsible for building the LCB adjacency lists and for connecting group 4 to group 5. It gets this data from the same file that was used by GETNAM. GETREF will terminate SEG3 if an early end of file is found

or if there are too many references to LCBs.

GETSON. This routine builds the ROM adjacency lists that define the edges of the call graph. Its input is the fourth record of a load map. It is possible that this record will begin with "UNSATISFIED EXTERNAL REFERENCE" error message from the loader. These messages are ignored.

All edges that lead to the root segment are ignored. References of this type are very common in FORTRAN programs. Unless the SYSEDIT option is used when compiling a FORTRAN program, all I/O will reference the file information tables in the main program. Therefore, any subroutine or function that performs an I/O operation will make a reference back to the main program.

If the ROMs are placed in more than one level, then all calls between levels are deleted.

GETSON will terminate SEG3 if there are too many edges in the call graph.

GETERR. This module is responsible for placing all data preset errors into the data base. These errors are read in from the third record of a load map. SEG3 is terminated if too many errors are found.

ANALYZ. ANALYZ insures that the call graph is transformed into a reduced segmentation forest. All ROMs and LCBs must be positioned properly as stated in Requirement 1.

POSMOD. This module uses Algorithm 3 to insure that all ROMs are positioned properly to avoid memory conflicts. This is done by breaking each level of the call graph down

into levels of forests.

PSUEDG. This routine adds pseudo edges to the level specified by the input parameter LEVEL. SEG3 will terminate if the pseudo root for level LEVEL cannot be found or if the additional edges result in too many total edges for the data base to hold.

CLASS and SEARCH. CLASS and SEARCH implement Algorithm 2. CLASS performs the necessary initialization for SEARCH. In particular, SEARCH marks all ROMs that must be moved to another level by setting MOVMOD to 4 and assigns FATHERs to all unmarked ROMs.

CLASS has one input parameter, S, which is the index of the pseudo root of the level to be searched. SEARCH has two input parameters. The first, V, is the vertex in the call graph that the OFS last visited and from which an edge will be selected to visit another vertex. The second parameter, RSEARCH, is the address of the entry point to SEARCH. This address is necessary for the recursion of SEARCH.

The fact that SEARCH is a recursive routine and that FORTRAN does not support recursion causes some problems. First, the FORTRAN compiler must be fooled into thinking that SEARCH does not call itself. This is done by passing SEARCH its own address, RSEARCH. Then, the entry point SEARCH and the parameter RSEARCH have the same address. If RSEARCH is declared as an external variable in the calling program, then it may be called by SEARCH.

A second problem is that FORTRAN does not have provisions

for automatically stacking and unstacking the local variables. Therefore, two stack operations, PUSH and POP, are used to implement a stack. The local variables that are stacked are RSEARCH; V and W, the end points of the edge just traversed; and LINK, the pointer to the next vertex to be searched. The variables that are not explicitly pushed onto the stack retain their values on the subsequent recursive calls.

To avoid problems in implementing this recursion, all variables in SEARCH that are not pushed onto the stack and are not part of the data base are explicitly made global variables by placing them in a LCB named /SEARCH/. These variables are initialized in CLASS and passed to SEARCH. These variables include N, M, NUMBER, and SNUMBER which were defined for Algorithm 2. Also in /SEARCH/ are the variables STACK, NSTACK, and MSTACK which are necessary to maintain the stack. STACK is an array in which the local variables are stacked. NSTACK is the number of variables in the stack at any time. MSTACK is the maximum number of variables allowed in the stack.

A final problem with recursion is that FORTRAN passes parameters by location. The specific problem here is that the variable W is input to the recursive routine SEARCH and the variable V in the next level of recursion is assigned this address. This results in the variables V and W having the same address after the first recursive call. To overcome this a dummy variable is used to pass the value of W.

V and the dummy variable then have the same address which does not matter.

An abbreviated version of the actual FORTRAN code of CLASS and SEARCH is listed below to show how the recursion is implemented.

```
SUBROUTINE CLASS(S)
EXTERNAL SEARCH
COMMON/SEARCH/M,N,NUMBER,SNUMBER,STACK,NSTACK,MSTACK
    .
    .
    .
CALL SEARCH(S,SEARCH)
RETURN
END
SUBROUTINE SEARCH(V,RSEARCH)
EXTERNAL RSEARCH
COMMON/SEARCH/M,N,NUMBER,SNUMBER,STACK,NSTACK,MSTACK
    .
    .
    .
CALL PUSH the local variables
DUMMY=W
CALL RSEARCH(DUMMY,RSEARCH)
CALL POP the local variables
    .
    .
    .
RETURN
END
```

MRKDEC. SEARCH found all of the lattice vertices in the call graph and deleted all fronds and reverse fronds. The ROMs in the data base that correspond to these vertices were marked by SEARCH to signify that these ROMs need to be moved to a new segmentation level. The function of MRKDEC is to find all of the descendants of these marked ROMs and to mark the descendants. To do this, the MOVMOD of the descendants is set to 5.

MRKDEC has one output parameter, CHANGE, which is used to tell POSMOD that no ROMs were marked by SEARCH and that

the call graph being processed has been transformed into levels of forests.

MOVMRK. After SEARCH and MRKDEC have determined which ROMs need to be moved to a new level, MOVMRK moves these ROMs, deletes the edges between levels, and partially reinitializes the moved ROMs. The input parameter LEVEL is used to determine the level that was just searched.

The following actions must be taken to move the ROMs to a new level. First, the new level is created by adding a new pseudo root to the data base. This increments the value of NOLEVEL. Then, the MODLEV of each marked ROM is set to the new value of NOLEVEL. The MOVMOD and FATHER of each marked ROM are re-initialized to allow another search to be performed. Then, the edges connecting the new level to the old levels are deleted and the proper pseudo edges are added to the data base.

POSCOM. POSCOM insures that all LCBs are positioned in their NCA and that the LCBs and the ROMs that preset data in them are placed in the same segment. This is done using Algorithm 6.

DIFLEV. This routine checks to see if any LCBs are referenced by ROM, from more than one level and repositions these ROMs in the root segment.

ANCSTR. The function of this routine is to position all LCBs not moved by DIFLEV into their NCA. Algorithm 5 is used to do this.

PRESET. The function of this routine is to rearrange the

ROMs to prevent data preset errors. Steps 10 through 21 of Algorithm 6 are used to do this. CHANGE is an output parameter that is used to tell POSCOM that no ROMs were moved.

REDUCE. This routine insures that the reduction rules described in Chapter IV are applied to each level.

ONESON. This routine implements reduction Rule 1 using the method outlined in the previous chapter.

EVENUP. The function of this routine is to implement reduction Rule 2. This routine was not implemented.

GENERT. This module is responsible for writing all of the segmentation directives requested by the user to the proper output file. For each level in the segmentation graph, the GLOBAL, INCLUDE, and TREE directives are written, followed by a single LEVEL directive to separate the levels. After all of the levels have been processed, an END directive is added to close the directive file. The techniques described in the previous chapter are used by the three following routines to create the segmentation directives.

GLOBAL. GLOBAL writes one saved GLOBAL directive to the output directive file for each LCB in the level being processed. The input parameter, LEVEL, specifies this level.

INCLUD. This routine generates one INCLUDE directive for each ROM in the level specified by the input parameter LEVEL. No directives are generated for pseudo roots or for ROMs that were not visited during the DFS.

TREE. This routine generates one TREE directive for each segment in the level specified by the input parameter

LEVEL. Each directive may be of arbitrary length so they are generated internally first as a string of characters and then output 72 characters at a time. All continuation cards have the continuation character "," in column one. This reduces the number of string characters per card to 71 for all continuation cards.

FNDCOM. This function returns the index of the LCB stored in /COMMON/ of group 4 that matches the input parameter NAME. Zero is returned if NAME is not found.

FNDMOD. This function returns the index of the ROM stored in /MODULE/ of group 2 that matches the input parameter NAME. Zero is returned if NAME is not found.

MOVEUP. Two segments, SON and DAD, are supplied to MOVEUP as input parameters. The function of MOVEUP is to move all of the ROMs and LCBs in SON to DAD and then to make all segments that are sons of SON, sons of DAD. This process is described in Algorithm 7.

NEARER. This function implements Algorithm 4 which finds the NCA of any two segments in the segmentation graph. These two segments, V and W, are the input parameters and are assumed to be in the same level.

POP. This routine returns the top value of the stack supplied to POP as parameters. This stack is defined by the input parameters STACK, the array containing the stacked values, and NSTACK, the number of variables in the stack. The output parameter VAR is used to return the top value. POP will terminate SEG3 if a stack underflow occurs, i.e.,

if NSTACK is less than or equal to zero upon entry to POP.

PRINT. This is a diagnostic routine that prints the entire data base on a default file named OUTPUT in a readable format. The input parameter TITLE is a single word used to identify the printout.

PSEUDO. This function returns the name of the pseudo root for the level specified by the input parameter LEVEL. The name that is produced by PSEUDO is *PSU*nn where nn is the level with leading zero.

PSUROT. PSUROT adds a pseudo root to the data base. To do this, a new level is created by incrementing NOLEVEL. Then a new pseudo root name is generated by PSEUDO and this new ROM is placed in the new level by setting its MODLEV to NOLEVEL. If this new ROM causes a ROM overflow in the data base, then SEG3 is terminated.

PUSH. This routine adds a value to a specified stack. The value is the input parameter VAR. The stack is defined by the input parameters STACK, NSTACK, and MSTACK. STACK and NSTACK are the same as in POP. MSTACK is the maximum number of values allowed in the stack. SEG3 is terminated if NSTACK becomes greater than MSTACK.

TIMER. This is a diagnostic routine that prints the input parameter TITLE, the date, the time of day, and the elapsed CPU time since the last call to TIMER. This information is written on a file named OUTPUT.

General Coding Techniques

This section discusses several coding conventions that

were used in writing SEG0, SEG1, SEG2, and SEG3. As mentioned earlier, the data base is made up of LCBs with one variable per LCB. In addition, the LCB name and the variable name are the same.

Another technique is to improve the diagnostic capabilities of the FORTRAN compiler by declaring all local variables in each module to be implicitly LOGICAL. Then the local variables that are used by each module are explicitly declared. By doing this, nearly all misspellings of variable names result in syntax errors and are easily detected.

A final convention is to reduce the number of GOTO statements used in each module. In addition, all GOTOs are in a forward direction and all statement labels are on CONTINUE statements.

VI. Conclusions

SEGMENT was implemented as described in the previous chapter. This chapter will discuss the results obtained from a limited testing of SEGMENT and will evaluate it against the requirements of Chapter III. Recommendations for further studies in this area will be given including improvements to SEGMENT and new approaches for determining the segmentation forest. Finally, a list of techniques for designing segmentable programs will be outlined.

Results

SEGMENT was tested against several FORTRAN programs to determine the feasibility of the algorithms used. These tests show that the basic concepts of SEGMENT are valid. The results of these tests will now be compared to the requirements in the discussion below.

The first requirement is that both the segmented and the unsegmented versions of the same program must produce the same results. For all of the programs tested, this was found to be true if the correct method of building the call graph was used. The test programs included cases of cycles, redundant edges, and lattice structures and all of these were handled properly by the DFS. In addition, all LCBs were positioned in the NCA segment of all of the segments that referenced each LCB, and all ROMs which preset data in LCBs were placed in the segment containing the LCB that was

preset by each ROM.

The second requirement is that the directives must significantly reduce the amount of central memory required to load and execute most programs without excessive use of other system resources. The main problem with analyzing this requirement is defining what a significant reduction is. In most cases, the user will have a goal for the amount of central memory that he would like his program to use. Meeting this goal is what is significant to the user. In addition, some programs are structured in such a way that virtually no memory savings can be made by segmenting the program. Therefore, whether or not a particular central memory savings is significant is highly dependent on the goals of the user and the program being segmented.

Typical memory reductions achieved by SEGMENT run in the range of 20% at a cost of a 20% increase in execution time. Also, the absolute file of segments for all of the test cases contains only one copy of each ROM. This means that the absolute file is about as small as possible. However, the 20% figure mentioned above can be somewhat misleading.

The amount of central memory that is necessary for a segmented or an unsegmented program is either the central memory needed to load the program or the central memory needed to execute the program, whichever is greater. Since the loader used to load a relocatable unsegmented program is much larger than the loader used to load an absolute segmented program, about half of the 20% memory savings is due

to a difference in loader size and will be saved no matter how the program is segmented. This is a very real savings, as it helps the user to reach his goal.

An experienced programmer can do better than the 10% actual reduction made by SEGMENT but not very much better. Therefore, in order to determine whether or not a 10% savings of central memory is significant, each user must also compare the small additional savings that could be made by manual segmentation with the effort needed to achieve this savings.

The third requirement is that all of the ROMs of the input program should be segmented. The fact that many of the ROMs in the system libraries contain indirect references to externals has prevented SEGMENT from completely meeting this requirement. Since SEGMENT ignores all ROMs in these libraries that do not contain indirect references are not segmented when they could possibly be segmented in such a way as to save more memory.

The fourth requirement is that the processor should be easy to use. Since the user must specify a method for building the call graph and the files and libraries that contain indirect references, SEGMENT is not as easy to use as it could be.

Recommendations

The results of this study can be used as a foundation for improving SEGMENT and for more sophisticated attempts of determining the segmentation forest of the input program.

The following recommendations are made for improving SEGMENT. First, an algorithm for applying reduction Rule 2 is needed. This rule states that two conflicting segments can be combined into a single segment if this combination does not increase the central memory needed to execute the program.

Second, a formal, more thorough test plan is needed. This will not be easy to do. For example, how is a program written so that its call graph will cause a specific execution path in SEGMENT to be tested?

A third recommendation for SEGMENT is to isolate the SEG3 data base by the use of abstract data types (Ref 10). This would combine the data base and all means of accessing and modifying it into a single module.

A fourth recommendation is to change the methods of building the call graph. Instead of specifying libraries and files with indirect references, only the ROMs that actually make the indirect references should be specified to be deleted from the call graph. Also, a set of these ROMs on the FORTRAN and SYSIO libraries should, by default, be deleted from the call graph unless otherwise directed by the user. This would allow most FORTRAN programs to be segmented with the user required to provide only the partial load sequence and the name of the root ROM.

The following recommendations are made for improvements beyond SEGMENT. The main problem with SEGMENT is that it does not minimize the use of memory. This is largely due

to the fact that it uses more than one level to express the segmentation forest. The following are two methods of generating a segmentation tree out of all of the vertices of the call graph.

The first method is to create a fully expanded tree out of the call graph and, then, to apply a set of reduction rules to the tree. Before a fully expanded tree can be created, all cycles and redundant edges are found and deleted and all lattice vertices are marked using a DFS. Then, each lattice vertex and all of its descendants are detached from the call graph and a separate copy of this subgraph is connected to each vertex that called that lattice vertex. The fully expanded tree will be much larger than the original call graph because of the large number of redundant vertices that are present. After the tree is created, then the reduction rules can be applied to it. These rules should include the two rules described in Chapter IV.

The fully expanded tree minimizes the amount of memory required to execute a particular program since each path from the root vertex to a leaf of the tree corresponds directly to the active ROMs that would be on the execution stack of the program when that leaf is in execution (see Chapter II). The biggest problem with this method is the size of the data structure needed to store the fully expanded tree which might have tens of thousands of vertices.

The second method of transforming a call graph into a segmentation tree is to move all lattice vertices, and

their descendants, to the NCA vertex of the vertices that call that lattice vertex. Again a DFS would be done first to remove cycles and redundant edges from the call graph and to mark the lattice vertices. This method does not minimize memory usage but it would be much easier to implement than the fully expanded tree method. Reduction rules could also be applied to this tree.

A second problem with SEGMENT is that it ignores the frequency with which each ROM is called. A method of assigning frequency of call weights to each vertex in the call graph would be very useful. This would require an additional source of input data used to build the call graph as the load map does not contain frequency information.

Third, it has been suggested that there is a relation between building the segmentation forest and the problem of how to pack ROMs together in a paged virtual memory system so as to minimize page faults (Ref 11). There are a large number of articles dealing with this second topic and these articles should be analyzed to try to determine this relation.

One final recommendation is to develop a set of design techniques that would allow a program to be segmented more effectively. A partial set of these techniques might be:

(1) The program should be structured so that all lattice vertices are leaves on the call graph.

(2) The program should not preset data in LCBs nor contain any indirect references to externals!

(3) The program should not use blank common; it cannot be segmented.

(4) Large data structures should be stored similarly to the data base of SEG3 with one variable name per LCB. This allows each piece of data to be placed in the NCA of all of the segments that use it and save more memory. However, this may increase execution time quite a bit.

(5) The program should use good structured design principles (Ref 12; Ref 13).

Summary

The purpose of this paper has been to describe a software processor that automatically generates segmentation directives for the loader on the CDC 6600. This processor, called SEGMENT, uses a depth-first search to position relocatable object modules into segments. The processor also positions labeled common blocks and handles data preset errors and indirect references. This processor is now up and running on the CDC CYBER 74 computer at the Aeronautical Systems Division, Wright-Patterson Air Force Base.

Bibliography

1. Control Data Corporation. Loader Version 1 Reference Manual, 60344200. Sunnyvale, California: Control Data Corporation, 1976.

2. Sayer, D. "Is Automatic Folding of Programs Efficient Enough to Displace Manual?" Communications of the ACM, 12:656-660 (December 1969).

3. Control Data Corporation, NOS/BE 1 Reference Manual, 60493800. Sunnyvale, California: Control Data Corporation, 1977.

4. Madnick, S. E. and J. J. Donovan. Operating Systems. New York: McGraw-Hill Book Co., 1974.

5. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Reading, Massachusetts: Addison-Wesley Publishing Co., 1976.

6. Tarjan, R. E. "Depth-first Search and Linear Graph Algorithms." SIAM Journel of Computing, 1:146-160 (June 1972).

7. Tarjan, R. E. "Finding Dominators in Directed Graphs." SIAM Journel of Computing, 3:62-89 (March 1974).

8. University of Washington. Catalogued Procedures and Other Techniques for Manipulating Control Cards, W00033. 1975.

9. Control Data Corporation. FORTRAN Extended Version 4 Reference Manual, 60305601. Sunnyvale, California: Control Data Corporation, 1976.

10. Parnas, D. L. "On the Criteria to be Used in Decomposing Systems Into Modules." Communications of the ACM, 15:1053-1055 (December 1972).

11. Gentleman, W. M., and J. I. Munro. "Designing Overlay Structures." Software-Practice and Experience, 7: 493-500 (1977).

12. Yourdon, E. and L. L. Constantine. Structured Design. New York: Yourdon, Inc., 1976.

13. Myers, G. J. Software Reliability-Principles and Practices. New York: John Wiley and Sons, 1976.

AD-A047 782 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
AUTOMATIC GENERATION OF SEGMENTATION DIRECTIVES ON THE CDC 6600--ETC(U)
DEC 77 S R SARNER
UNCLASSIFIED AFIT/GCS/MA/77D-4 NL

2 OF 2
AD
A047782

END
DATE
FILMED
1-78
DDC

Appendix A

Graph Theory Definitions

A directed graph $G = (V,E)$ is an ordered pair consisting of a set of vertices V and a set of edges E. Each edge is an ordered pair (v,w) of distinct vertices. A graph $G_1 = (V_1,E_1)$ is a subgraph of a graph $G_2 = (V_2,E_2)$ if V_1 V_2 and E_1 E_2. If (v,w) is an edge in E, then vertex w is said to be adjacent to vertex v. The adjacency list for vertex v is a list of all vertices w adjacent to v. A graph can be represented by /V/ adjacency lists, where /V/ is the number of vertices in G. The edge (v,w) is said to go from v to w. The number of vertices adjacent to v is called the out-degree of v. The number of vertices that w is adjacent to is called the in-degree of w.

A path $p(v_1,v_n)$ is a sequence of edges of the form $(v_1,v_2),(v_2,v_3),\ldots,(v_{n-1},v_n)$. A path from v_1 to v_n is of length n-1. A vertex w is reachable from vertex v if there is a path p(v,w). A path is simple if all of its edges are distinct. A cycle is a simple path of length at least one which begins and ends at the same vertex. A directed graph is acyclic if it contains no cycles.

A rooted, directed tree T is a graph with one distinguishing vertex r, called the root, such that every vertex in T is reachable from r, no edges enter r, and exactly one edge enters every other vertex in T. A directed graph consisting of a collection of trees is called a forest.

Let $F = (V,E)$ be a graph which is a forest. If (v,w) is in E, then v is called the father of w and w is the son of v. If there is a path $p(v,w)$, then v is an ancestor of w and w is a descendant of v. Every vertex is defined to be an ancestor and a descendant of itself. If $v \neq w$, then v is a proper ancestor of w and w is a proper descendant of v. A vertex with no proper descendants is called a leaf. If T_1 is a tree and is a subgraph of tree T_2, then T_1 is a subtree of T_2. If T is a tree which is a subgraph of a directed graph G and T contains all the vertices of G, then T is a spanning tree of G.

The height of a vertex v in a tree is the length of the longest path from v to a leaf. The depth of a vertex v in a tree is the length of the path $p(r,v)$. The height of a tree is the height of the root. The level of a vertex v in a tree is the height of the tree minus the height of v.

Let V_1 be a set of one or more vertices of a tree T. Let A be the largest subset of V such that each element of A is an ancestor of all elements of V_1. A is called the set of common ancestors of V_1. A always contains r, the root of T. The nearest common ancestor of V_1 is the vertex in A at the highest level. Let V_2 be a set of more than one vertices in more than one disjoint trees of a forest with one tree designated as the primary tree. The nearest common ancestor of the elements in V_2 is the root of the primary tree.

Appendix B

SEGMENT User's Guide

SEGMENT is a processor that automatically generates segmentation directives. Segmentation can be used to reduce the amount of central memory required to load and execute most programs. This may allow a large program to be run on Intercom or to be run with a higher priority on Batch. SEGMENT may be used on Intercom if the program being segmented requires less than 56600_8 words to load normally. This allows the user to learn how to use SEGMENT at the terminal and then apply SEGMENT to a larger program on Batch.

SEGMENT is a Control Card Language procedure. It is stored on a permanent file named SEGMENT,ID=AFIT and may be used with the following syntax.

BEGIN,SEGMENT,lfn_1,lfn_2,lfn_3,lfn_4,lfn_5,lfn_6,lfn_7.

lfn_1 is the local file name under which SEGMENT is attached. The default name for lfn_1 is PROFIL.

lfn_2 is a file containing a partial load sequence. Only LOAD, SLOAD, LIBLOAD, and LDSET control statements may be included in this sequence. All LDSET options except MAP and ERR may be used. All files used in this sequence must exist prior to the BEGIN statement. If lfn_2 is not specified, then the default name INPUT is assumed.

lfn_3 is a file containing the user's directives to

SEGMENT. These directives specify the name of the program being segmented, an entry point in that program, and one of four ways to segment that program. The default name for lfn_3 is INPUT. When lfn_2 and lfn_3 are on the same file, lfn_2 must precede lfn_3.

lfn_4 is the file to which the absolute segments are written by the processor. This file can be catalogued (if it was requested as a permanent file prior to the BEGIN statement) or executed with a name call statement. The default name for this file is ABS.

lfn_5 is the file to which the Segmentation directives are written by the processor. The default name for this file is PUNCH.

lfn_6 is the file to which the load map of the SEGLOAD that created the absolute file (lfn_4) is written by the processor. The default name for this file is OUTPUT.

lfn_7 is the file to which the data base used by SEGMENT is written. This file provides diagnostic information about how the program was segmented and can be used to do further manual segmentation of the directives produced by SEGMENT. The default name for this file is OUTPUT.

The first two directives on lfn_3 tell SEGMENT the name

of the program and the name of an entry point in that program at which execution will begin. The third and all subsequent directives instruct SEGMENT to use one of four methods to analyze the program. One of these methods is selected by making the third directive either METHOD1, METHOD2, METHOD3, or METHOD4. These four methods are necessary because it is possible that one of the relocatable object modules in the user's program (which includes all modules used from libraries) may reference another module in such a way that the loader does not detect the reference. These references are called indirect references and may cause the segmented program not to execute correctly. When the user knows of an indirect reference in his program (see the Loader Version 1 manual, page 5-5), he can insure that the indirect reference will not affect his program by directing SEGMENT to use the correct method.

METHOD1 is used when no indirect references are in the user's program or when he knows that the indirect references that are there will not affect the program. All lines following a METHOD1 directive on lfn_3 are ignored.

METHOD2 is used to handle indirect references. The METHOD2 directive must be followed by all of the file and library names in the partial load sequence that contain indirect references. The system libraries FORTRAN and SYSIO both contain indirect references.

METHOD3 is used for very large programs that contain no indirect references should METHOD1 fail to complete execution

because of the size of the program. All lines following a METHOD3 directive on lfn_3 are ignored.

It is possible that when METHOD2 is used, part of the user's program will not be segmented. METHOD4 can be used to correct this situation. METHOD4 requires the same file and library names as METHOD2.

All directives in lfn_3 must begin in column 1 and may not contain more than seven characters. All directives, except the name of the program, have default values if they are not explicitly specified. The entry point name defaults to the program name. The method defaults to METHOD2 with the names FORTRAN and SYSIO. The entry point directive must be included if a method is specified explicitly.

The two input files, lfn_2 and lfn_3, and the two output files, lfn_6 and lfn_7, are not rewound by SEGMENT. The files lfn_1, lfn_2 and lfn_3 are automatically rewound before being used by SEGMENT.

The amount of resources that SEGMENT requires varies greatly with the program being segmented. As a guide, request enough central memory to do a basic load of the program plus 2000_8 words. To begin with, request I/O and CPU time limits of IO100 and T30. These may then be modified as necessary.

The following are examples of deck set-ups for various jobs using SEGMENT.

Example 1

Compile a FORTRAN source program called SMPL, segment SMPL, and then execute it with a deck of data. METHOD2 is used by default and the FORTRAN and SYSIO libraries contain indirect references.

```
S11,CM125000,T30,I0100,STCSB.T770145,SARNER
FTN.
ATTACH,PROFIL,SEGMENT,ID=AFIT,SN=AFIT.
BEGIN,SEGMENT.
ABS.
(7/8/9)
    (FORTRAN source program)
(7/8/9)
LOAD,LGO.
(7/8/9)
SMPL
(7/8/9)
    (Data deck)
(6/7/8/9)
```

Example 2

Segments a large number of relocatable object modules from several files. The user knows that the fifth relocatable object module on the file MAIN passes an external variable as a parameter to the tenth relocatable object module of the file SUBS. This is one example of an indirect reference. New files are created so that these two relocatable object modules may be placed in a file that will not be segmented. The absolute file is saved as a permanent file. No directives or load map is needed so they are written to a dummy file name.

```
S22,T40,IO150,CM175000,STCSB.T770145,SARNER
ATTACH,MAIN,MAINPROGRAM.
ATTACH,LIB1,LIBRARY1.
ATTACH,LIB2,LIBRARY2.
ATTACH,LIB3,LIBRARY3.
ATTACH,SUBS,SUBROUTINES.
COPYBR,MAIN,FILE1,4.
COPYBR,MAIN,FILE2,1.
COPYBF,MAIN,FILE1.
SKIPF,SUBS,9.
COPYBR,SUBS,FILE2,1.
LIBRARY,LIB3.
REQUEST,ABS,*PF.
ATTACH,SEG,SEGMENT,ID=AFIT,SN=AFIT.
BEGIN,SEGMENT,SEG,,,,DUMMY,DUMMY,DUMMY.
RETURN,DUMMY.
CATALOG,ABS,ABSFILE.
(7/8/9)
LOAD(FILE1)
LOAD(FILE2)
LDSET(LIB=LIB1/LIB2,PRESET=INDEF)
SLOAD(SUBS,SUB1,SUB2)
(7/8/9)
MAINAME
MAINAME
METHOD2
FORTRAN
SYSIO
FILE2
(6/7/8/9)
```

Appendix C

Sample Segmented Load Map

This appendix gives the first page of each record of a segmented load map. The first record shows that ROMs were used from one file, named A, and from two libraries, named Z and FORTRAN. The information that is taken off of this record are the ROM names and lengths and the LCB names and lengths references by each ROM. For example, the ROM named PLOTS taken from the library named Z is 113_8 words long and references the LCBs named QQPINS and QQPINN which are 7_8 words long and 2_8 words long respectively. The second record is empty and is not shown here.

The third record lists all possible significant data preset errors. For example, in the first line of the record the ROM named ADDFIL presets data in LCB named PLFILE.

The fourth record lists all ROM references to other ROMs. For example, the first line says that the ROM named OUTF= references the ROM named FORSYS=. The second line of this record says that the ROM named GOTOER= also references ROM named FORSYS=.

```
READING FILE        A
         PROGRAM   GRAPH    L=007104  BLOCKS           CM035275
         PROGRAM   PTNT     L=000173  BLOCKS           CM035275
         PROGRAM   VGRAPH   L=000410  BLOCKS           CM035275
         PROGRAM   VARPT    L=000475  BLOCKS           CM035275
CONSULTING LIBRARY 7
         PROGRAM   PLOTS    L=000113  BLOCKS  QQPINS   CM000007  QQPINN   CM000002
         PROGRAM   PLOT     L=000020
         PROGRAM   ADDFIL   L=000036  BLOCKS  PLFILE   CM000041
         PROGRAM   COMPRS   L=000011
         PROGRAM   DONEPL   L=000024
         PROGRAM   QQPLUS   L=000623  BLOCKS  QQPINT   CM002026  QQPINS   CM000007  QQSIA    CM000002  QQPINN   CM000002
         PROGRAM   PLTTP    L=000072
         PROGRAM   QQCPRS   L=000053
         PROGRAM   QQPLUT   L=000214  BLOCKS  QQPLNT   CM000012
         PROGRAM   QQLOAD   L=000250
         PROGRAM   IQQQ6    L=000025
         PROGRAM   LQQBUF   L=000200  BLOCKS  QQPINT   CM002026  QQPINS   CM000007
         PROGRAM   QQINIP   L=000046  BLOCKS  QQSIA    CM000002  QQPINT   CM002026  QQPINS   CM000007
         PROGRAM   QQQRCH   L=000022
         PROGRAM   BGNPL    L=001155  BLOCKS  QQPLOT   CM000074  QQI9I    CM000050  QQI7I    CM000072  QQI7I2   CM000043
                                      BLOCKS  QQBLK1   CM000005  QQLEGD   CM000013  QQALFD   CM000004  QQALFP   CM000014
                                      BLOCKS  QQAPP    CM000002  QQS1S    CM000070  QQMPR2   CM000011  QQANGK   CM000024
         PROGRAM   QQERRW   L=000046
         PROGRAM   QQPTCT   L=000014  BLOCKS  QQCOM    CM000001  QQPLNT   CM000012  QQAPP    CM000002  QQSIA    CM000002
         PROGRAM   QQEPR    L=000077
         PROGRAM   QQROUT   L=000155  BLOCKS  QQPINT   CM002026  QQPINS   CM000007
         PROGRAM   QQPK8    L=000043
         PROGRAM   QQQEL    L=000065
         PROGRAM   QQPEND   L=000077  BLOCKS  QQSIA    CM000002
         PROGRAM   IQQERG   L=000014
         PROGRAM   IQQJ1P   L=000133  BLOCKS  QQDAT    CM000057
         PROGRAM   QQCPTL   L=000020  BLOCKS  QQCOM    CM000001
         PROGRAM   IQQEOF   L=000052  BLOCKS  QQENTP   CM000017
         PROGRAM   IQQJOB   L=000767  BLOCKS  QQDAT    CM000057
         PROGRAM   IQQSLU   L=000120  BLOCKS  QQENTP   CM000017  QQLUST   CM000007
         PROGRAM   QQLYNN   L=000034
         PROGRAM   QQMVC    L=000070  BLOCKS  QQDAT    CM000057
         PROGRAM   QQRSTL   L=000011  BLOCKS  QQFLGS   CM000051
         PROGRAM   QQSYSP   L=000010
         PROGRAM   QQSYSP   L=000010
         PROGRAM   IQQSTS   L=000033  BLOCKS  QQENTP   CM000017  QQLUST   CM000007
         PROGRAM   QQQJCD   L=000226
         PROGRAM   QQJOB    L=000025
         PROGRAM   QQGETIPF L=000126
         PROGRAM   QQTQEF   L=000174
         PROGRAM   QQREAD   L=000062  BLOCKS  QQENTP   CM000017
         PROGRAM   QQ3RST   L=000113  BLOCKS  QQP3D    CM000046  QQ3EX    CM000044  QQ4EX    CM000050
         PROGRAM   QQ3SET   L=000023  BLOCKS  QQ4EX    CM000050  QQP3D    CM000046  QQ3EK    CM000044
         PROGRAM   QQSTP    L=000040  BLOCKS  QQENTP   CM000017  QQLUST   CM000007
         PROGRAM   NAME     L=000012
         PROGRAM   QQSYST   L=000031
         PROGRAM   PERMFIL  L=000736
         PROGRAM   IQQPLA   L=000016
CONSULTING LIBRARY FORTRAN
         PROGRAM   FORSYS=  L=000601  BLOCKS  Q8.IO.   CM000131  FCL.C.   CM000023
         PROGRAM   OUTF=    L=000151  BLOCKS  Q8.IO.   CM000131  FCL.C.   CM000023
         PROGRAM   GOTOER=  L=000014
         PROGRAM   INPB=    L=000314  BLOCKS  IO.BUF.  CM000227  FCL.C.   CM000023
         PROGRAM   REWINB=  L=000037
         PROGRAM   Q8NTRY=  L=000000  BLOCKS  FCL.C.   CM000023  Q8.IO.   CM000131
```

```
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. ADDFIL   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PLFILE   IN ANOTHER SEGMENT
// PROG. QQPLUS   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QQPINS   IN ANOTHER SEGMENT
// PROG. QQPLUS   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QQPINS   IN ANOTHER SEGMENT
// PROG. IDDSLU   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QQENTP   IN ANOTHER SEGMENT
// PROG. IDDSLU   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QQENTP   IN ANOTHER SEGMENT
// PROG. IDDSLU   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QQLUST   IN ANOTHER SEGMENT
// PROG. QANTRY=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QA.IO.   IN ANOTHER SEGMENT
// PROG. QANTRY=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QA.IO.   IN ANOTHER SEGMENT
// PROG. QANTRY=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QA.IO.   IN ANOTHER SEGMENT
// PROG. QANTRY=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QA.IO.   IN ANOTHER SEGMENT
// PROG. QANTRY=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QA.IO.   IN ANOTHER SEGMENT
// PROG. QANTRY=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. QA.IO.   IN ANOTHER SEGMENT
// PROG. FORJTL=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. FCL.C.   IN ANOTHER SEGMENT
// PROG. FORJTL=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. FCL.C.   IN ANOTHER SEGMENT
// PROG. FORJTL=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. FCL.C.   IN ANOTHER SEGMENT
// PROG. FORJTL=  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. FCL.C.   IN ANOTHER SEGMENT
// PROG. OPEN.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. OPEN.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. JMPS.RM  IN ANOTHER SEGMENT
// PROG. OPEN.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. JMPS.RM  IN ANOTHER SEGMENT
// PROG. OPEN.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. JMPS.RM  IN ANOTHER SEGMENT
// PROG. OPEN.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. OPEN.FO  IN ANOTHER SEGMENT
// PROG. OPEN.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. OPEN.FO  IN ANOTHER SEGMENT
// PROG. OPEN.RM  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. OPEN.RM  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. OPEN.RM  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. OPEN.RM  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. MEMC.RM  IN ANOTHER SEGMENT
// PROG. OPEN.RM  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. OPES.FO  IN ANOTHER SEGMENT
// PROG. PUT.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. TERM.RM  IN ANOTHER SEGMENT
// PROG. PUT.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. PUT.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PUT.FO   IN ANOTHER SEGMENT
// PROG. PUT.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. PUT.FO   IN ANOTHER SEGMENT
// PROG. PUT.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. JMPS.RM  IN ANOTHER SEGMENT
// PROG. CLSE.RM  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. CLSE.FO  IN ANOTHER SEGMENT
// PROG. Z.SQ     , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. Z.SQ     , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.RT   IN ANOTHER SEGMENT
// PROG. Z.SQ     , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.RT   IN ANOTHER SEGMENT
// PROG. Z.SQ     , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.RT   IN ANOTHER SEGMENT
// PROG. SKEL.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. SKEL.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. SKEL.FO  IN ANOTHER SEGMENT
// PROG. SKEL.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. SKEL.FO  IN ANOTHER SEGMENT
// PROG. GET.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. GET.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. TERM.RM  IN ANOTHER SEGMENT
// PROG. GET.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.FO   IN ANOTHER SEGMENT
// PROG. GET.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.FO   IN ANOTHER SEGMENT
// PROG. GET.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.RT   IN ANOTHER SEGMENT
// PROG. GET.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.RT   IN ANOTHER SEGMENT
// PROG. W.SQ     , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. W.SQ     , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. GET.RT   IN ANOTHER SEGMENT
// PROG. REW.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. REW.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. REW.FO   IN ANOTHER SEGMENT
// PROG. REW.SQ   , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. REW.FO   IN ANOTHER SEGMENT
// PROG. WEOK.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. STRT.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL BL. AQR.RM   IN ANOTHER SEGMENT
// PROG. STRT.SQ  , SEGMENT  S2      TRIES TO LOAD INTO GLOBAL
```

ABNORM.	GRAPH	FORSYS=	+021511	GRAPH	OUTF=	+022362					
					GOTOER=	+022412					
ADDFIL	GRAPH	ADDFIL	+011544	GRAPH	PLOTS	+011415					
ALOG10.	GRAPH	ALOG	+032102	GRAPH	NUMBER	+030350	+030375				
ALOG.	GRAPH	ALOG	+032062	GRAPH	XTOY=	+032153					
AMAD.SQ	GRAPH	GET.SQ	+036356	GRAPH	A.SQ	+037151					
					BTRT.SQ	+037347	+037362	+037406	+037407	+037416	+037417
						+037420	+037426	+037434	+037437	+037445	
					ESU.SQ	+041437	+041501	+041526			
ANBL.SQ	GRAPH	GET.SQ	+036352	GRAPH	BTRT.SQ	+037361	+037377	+037404	+037462		
AXIS	GRAPH	AXIS	+023467	GRAPH	VGRAPH	+010411	+010417				
					VARPT	+011341	+011044				
BEN.	GRAPH	FORUTL=	+025515	GRAPH	FORSYS=	+021562	+021744				
BGNPL	GRAPH	BGNPL	+013640	GRAPH	PLTTR	+012506					
BUFIN.	GRAPH	BUFIN=	+025332	GRAPH	DDREAD	+020214					
BUFID.	GRAPH	BUFIO=	+031333	GRAPH	BUFIN=	+025340					
					BUFOUT=	+025406					
BUFOUT.	GRAPH	BUFOUT=	+025366	GRAPH	DDREAD	+020205					
CBD.	GRAPH	FORUTL=	+025610	GRAPH	FORSYS=	+021447	+021570	+021670	+021771		
					INCOM=	+022232					
CHEK.RM	GRAPH	CHEK.RM	+037621	GRAPH	UNIT	+021462					
CHHR.SQ	GRAPH	CHHR.SQ	+041336	GRAPH	OPEN.SQ	+033563					
					PUT.SQ	+034570					
					CHEK.RM	+037616					
CLSF.RM	GRAPH	CLSF.RM	+035656	GRAPH	FORSYS=	+021547	+021711				
					OUTF=	+022261					
					OUTC=	+023047					
					BUFOUT=	+025400					
CLSV.SQ	GRAPH	CLSV.SQ	+041174	GRAPH	PUT.SQ	+035054	+035244				
					GET.SQ	+037030					
COMPRS	GRAPH	COMPRS	+011600	GRAPH	PLOTS	+011416					
COS	GRAPH	SINCOS=	+030222	GRAPH	SYMBOL	+024127					
COS.	GRAPH	SINCOS=	+030230	GRAPH	AXIS	+023540					
DATE	GRAPH	CLOCK=	+025271	GRAPH	GOJOB	+017616					
DONEPL	GRAPH	DONEPL	+011611	GRAPH	PLOTS	+011476					
DXIT.SQ	GRAPH	GET.SQ	+037357	GRAPH	SKFL.SQ	+036320					
					W.SQ	+037120	+037122				
ENCODI.	GRAPH	ENCODE=	+025150	GRAPH	DOPEND	+015572					
					DOLYNN	+017135					
ERR.RM	GRAPH	ERR.RM	+040123	GRAPH	OPEN.SQ	+033627	+033631	+033534	+033675	+033742	+033744
						+033776	+034026	+034077	+042503	+042504	+042505
						+042506	+042507	+042510	+042511	+042513	+042514
						+042515	+042516	+042517	+042520	+042522	
					OPEN.RM	+034037	+034127	+034174	+034252	+034254	+034260
						+034262	+034263	+034264	+042477	+042500	+042501
						+042502					
					PUT.SQ	+034273	+034300	+034334	+034355	+034360	+034365
						+034434	+034476	+034532	+034561	+035001	+035035
						+035031	+035051	+035572	+035500	+042467	+042470
						+042471	+042472	+042473	+042474	+042475	
					CLSF.RM	+035660	+035671	+035677	+042460	+042461	+042462
						+042463	+042464	+042465	+042466		
					SKFL.SQ	+036004	+036037	+036012	+042440	+042441	+042442
						+042443	+042444	+042445	+042446		
					GET.SQ	+036051	+035755	+036072	+036101	+036125	+036615
						+036735	+036770	+037013	+042431	+042432	+042433
						+042434	+042435	+042436	+042437		
					W.SQ	+037153					
					REW.SQ	+037156	+042415	+042416	+042417	+042420	+042421

Vita

Steven R. Sarner was born on 3 March 1948 in Minneapolis, Minnesota. He graduated from Hopkins High School, Hopkins, Minnesota in 1966. He graduated from the United States Air Force Academy with a Bachelor of Science degree in Aeronautical Engineering and was commissioned on 3 June 1970. After graduation from Undergraduate Pilot Training in 1971, he was assigned to Norton AFB, California. There, he flew the C-141 until coming to the Air Force Institute of Technology in June 1976. He attended Squadron Officer's School in residence during 1975.

Permanent Address: 3610 Ridgewater Trail
Marietta, Georgia 30062

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER GCS/MA/77D-4	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Automatic Generation of Segmentation Directives on the CDC 6600 Computer		5. TYPE OF REPORT & PERIOD COVERED M.S. Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Steven R. Sarner Captain USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson Air Force Base, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Aeronautical Systems Division (ASD/ADDS) Wright-Patterson Air Force Base, Ohio 45433		12. REPORT DATE December 1977 13. NUMBER OF PAGES 99
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

Jerral F. Guess, Captain, USAF
Director of Information

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

CDC 6600	nearest common ancestor
segmentation	program call graph
overlays	cycle elimination
loader	memory management
depth-first search	

to-the-base-8

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Central memory is a critical resource on the Aeronautical Systems Division CDC 6600 computer system. This is especially true in the time-sharing environment where the standard maximum field length for a user's program is 60000_8 words of central memory. The amount of central memory required to load and execute a program may be reduced by using the CDC segmentation scheme in which a tree structure of relocatable (over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

(cont fr D 1473A)

object modules is specified which permits sharing of central memory. However, the design of an effective tree structure for segmenting a large application program can be very difficult and time consuming.

This thesis develops a software processor called SEGMENT that automatically generates segmentation directives for a user's program that describe a tree structure for that program. SEGMENT uses a depth-first search to determine the tree structure of relocatable object modules. SEGMENT also uses algorithms which correctly position labeled common blocks in this tree structure and which eliminate data preset errors and indirect references to externals.

A

1473B

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)